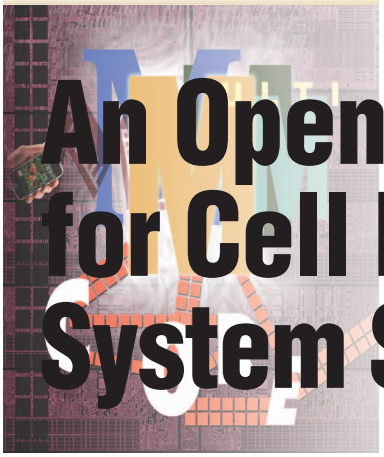

An Open Source Environment for Cell Broadband Engine System Software

Michael Gschwind, IBM T.J. Watson Research Center
David Erb, Sid Manning, and Mark Nutter, IBM Austin

The Cell Broadband Engine provides the first implementation of a chip multiprocessor with a significant number of general-purpose programmable cores targeting a broad set of workloads. Open source software played a critical role in the development of the Cell software stack.

Computer architects rarely introduce new architectures because incumbent architectures offer significant advantages due to tool maturity, programmer familiarity, and software availability. New architectures are usually a response to tectonic shifts in technology and market conditions. Thus, the original System/360 architecture was the first architecture to respond to mass production of systems. RISC systems corresponded to the introduction of VLSI manufacturing and the advent of single-chip microprocessors.

As the era of pure CMOS frequency scaling ends, architects must again respond to massive technological changes by more efficiently exploiting density scaling. The Cell Broadband Engine (Cell BE) answers these challenges by providing the first implementation of a chip multiprocessor with a significant number of general-purpose programmable cores targeting a broad set of workloads, including intensive multimedia and scientific processing.

Jointly developed beginning in 2000 by IBM, Sony, and Toshiba (STI) for the PlayStation 3 as well as other data-processing-intensive environments, Cell's design goal was to improve performance an order of magnitude over that of desktop systems shipping in 2005.[1-3] To meet that goal, designers had to optimize performance against area, power, volume, and cost in a manner not possible with legacy architectures. Thus, the design strategy was to exploit application parallelism through numerous cores that support established application models, thereby ensuring good programmability as well as programmer efficiency.[4]

The resulting Cell design is a heterogeneous, multicore chip capable of massive floating-point processing optimized for computation-intensive workloads and rich broadband media applications. As the "Cell BE Architecture Overview" sidebar describes, the design consists of one 64-bit Power processor element (PPE), eight accelerator processors called Synergistic Processor Elements (SPEs), a high-speed memory controller, a high-bandwidth element interconnect bus, and high-speed memory and I/O interfaces, all integrated on-chip.

SOFTWARE CHALLENGES

When we first outlined the Cell system's basic notions, we immediately realized that this revolutionary microprocessor design could substantially enhance application performance, but the task at hand was massive. Developing a new architecture has a set of risks that microprocessor design teams rarely face. Failure to verify that a new architecture responds to the needs that led to its conception, or to provide a satisfactory software stack to early adopters, usually will result in the failure of an architecture launch and its eventual demise.

In addition to the traditional challenge of defining a new microarchitecture, the design team faced the challenge of ensuring that the architecture can efficiently operate across a wide range of applications. Given the many innovations in Cell, it was important to provide early proof-of-concept to test and refine concepts that form the basis of the Cell BE Architecture (CBEA) as it is known today and its first implementation, the Cell Broadband Engine.

0018-9162/07/$25.00 © 2007 IEEE

Cell BE Architecture Overview

We created the Cell Broadband Engine Architecture (CBEA) to address the needs of applications as they embrace chip multiprocessing. Rather than merely replicating a core multiple times on a chip, the Cell's heterogeneous architecture offers a mix of execution elements optimized for a spectrum of functions. Applications execute on this system, rather than a collection of individual cores, by partitioning the application and executing each component on the most appropriate execution element. While supporting different execution elements, the architecture also ensures efficient data sharing by providing a common system view of addressing, data types, and system functions across the heterogeneous execution elements. Based on this common system view, a Cell BE application process can consist of threads (lightweight processes) on both types of processor elements.

As Figure A shows, the Cell Broadband Engine, the first implementation of the CBEA,[1] includes a Power Architecture processor and eight attached processor elements. An internal high-performance element interconnect bus integrates the processor elements.

With a clock speed of 3.2 GHz, the Cell processor has a theoretical peak performance of 204.8 Gflop/s (single precision) and 14.6 Gflop/s (double precision). The element interconnect bus supports a peak bandwidth of 204.8 Gbytes/s for intrachip data transfers, the memory interface controller provides a peak bandwidth of 25.6 Gbytes/s to main memory, and the I/O controller provides peak bandwidth of 25 Gbytes/s inbound and 35 Gbytes/s outbound.

Power Processor Element

The Power processor element (PPE) consists of a 64-bit, multithreaded Power Architecture processor with two concurrent hardware threads. The PPE supports the Power Architecture vector multimedia extensions to accelerate multimedia applications using SIMD execution units. The processor has a memory subsystem with separate first-level 32-Kbyte instruction and data caches, and a 512-Kbyte unified second-level cache. By using a Power Architecture processor as the base building block of the CBEA, we leveraged our decade-long experience with this mature and tuned architecture, as well as a stable software environment.

Synergistic Processor Element

The eight on-chip synergistic processor elements (SPEs) provide a significant portion of compute power in a Cell system.[2] An SPE consists of a new processor—the synergistic processor unit (SPU)—designed to accelerate a wide range of workloads by providing an efficient data-parallel architecture and the synergistic memory flow controller (MFC), providing coherent data transfers to and from system memory.

The SPU cannot access main memory directly; the SPU obtains instructions and data from its 256-Kbyte local store and it must issue DMA commands to the MFC to bring data into the local store or write results back to main memory. In parallel to MFC data transfers, the SPU processes data stored in its private local store.

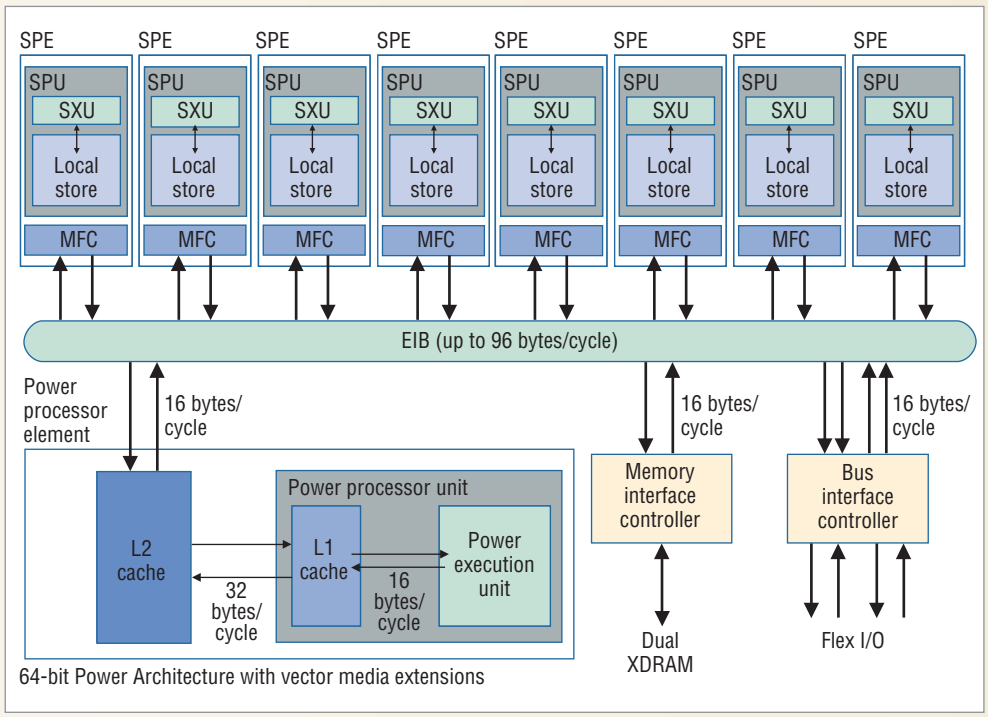

Figure A. Cell Broadband Engine system diagram. The system includes a Power Architecture processor and eight attached processor elements; an internal high-performance element interconnect bus integrates the processor elements.

The local store provides each SPU with private data access capability, guaranteed data availability, and deterministic access latency. The local store architecture offers logic simplicity, as cache-hit and coherence logic do not affect the critical memory access operations during load and store operations, allowing faster and more compact implementations. All data accesses with load and store operations refer directly to physical locations within an SPE's local store without further address translation.

Memory Flow Controller

To access global data shared between threads executing on the PPE and other SPEs, each SPE includes an MFC, which performs data transfers between SPU-local storage and system memory. The MFC provides the SPEs with access to system memory by supporting high-performance direct memory access (DMA) data transfer between the system memory and the local store. Data transfers can range in size from a single byte to 16-Kbyte blocks.

The MFC transfers copy between local store and system memory. An MFC transfer request specifies the local store location as the physical address in the local store. It specifies the system memory address as a Power Architecture virtual address, which the MFC's memory management logic translates to a physical address based on system-wide page tables that the Power Architecture specification provides.

Using the same virtual addresses to specify system memory locations independent of processor element type enables seamless data sharing between threads executing on both the PPE and SPE. An application executing on Cell can pass a PPE-generated pointer to code executing on the SPE and use it to specify the source or target in an MFC transfer request. Using full memory translation also ensures data protection between processes, as a thread can only access the system memory mapped into the associated process's virtual memory space.

Finally, using virtual addressing makes traditional operating system services such as demand paging available to SPE threads. When an SPE thread references paged-out memory via its associated MFC, the MFC's memory management unit generates a page-fault exception and delivers it to the PPE. The PPE then services the page fault on behalf of the SPE. When the page fault service has completed, the PPE restarts the MFC transfer that caused the page fault.

Memory Management

Multiple SPEs can share an address space with PPE threads in a Cell BE application, but at the same time other SPEs can reference different virtual memory spaces associated with respective applications executing concurrently in the system. To support this, each MFC includes a memory management unit (MMU) to provide address translation of system addresses in transfer requests. The MFC participates in the memory coherence protocols to ensure page table coherence.

Because each SPE contains an independent MMU, an SPE can execute independently from the PPE. However, the SPE is optimized for user-level data processing. Only the PPE performs privileged operations such as handling page faults, changing memory translation, and so forth, providing a centralized system control function. The Cell BE supports this by forwarding all exception-type events to the PPE via the on-chip interrupt controller.

Each MFC can be programmed to perform memory transfers either from the local SPU by placing commands in a 16-deep command queue using so-called SPU channel instructions or from remote nodes via memory-mapped I/O (MMIO). In addition to DMA transfers, the MFCs can also participate in the Power Architecture load-and-reserve and store-conditional lock synchronization and execute memory-synchronizing operations. Finally, the MFC supports list commands corresponding to an "MFC program" specifying a sequence of transfer requests.

Element Interconnect Bus

The element interconnect bus (EIB) provides high-bandwidth communication with a peak bandwidth of 204.8 Gbytes/s for intrachip data transfers among the PPE, the SPEs, and the memory and I/O interface controllers. The EIB has separate communication paths for commands (requests to transfer data to or from another element on the bus) and data. The EIB command path consists of a star-network to perform coherence actions. The EIB data network consists of four data rings—two rings running clockwise, two rings running counterclockwise.[3]

References

1. J. Kahle et al., "Introduction to the Cell Multiprocessor," *IBM J. Research and Development*, Sept. 2005, pp. 589-604.
2. M. Gschwind et al., "A Novel SIMD Architecture for the Cell Heterogeneous Chip Multiprocessor," *Hot Chips 17*, Aug. 2005.
3. S. Clark et al., "Cell Broadband Engine Interconnect and Memory Interface," *Hot Chips 17*, Aug. 2005.

DEVELOPING AN OPEN SOURCE STRATEGY

To succeed, modern technology solutions require rapid deployment in the marketplace. To address this challenge, the design team turned to open source software to accelerate the development of an ecosystem for the Cell architecture. Open source software allowed us to rapidly deploy an environment to be used both for architecture exploration and as an early adopter platform for the development of architecture verification suites, libraries, middleware, and sample applications.

The Cell open source software strategy had four phases:

- initial proof-of-concept focused on validating the design goals, compilation concepts, and programming paradigms developed in conjunction with the architecture definition;
- formative software phase supporting early adopter code for libraries, middleware, and applications;
- programming model innovation phase using a richer set of primitives, tools, and environments to explore the most efficient software development paradigms for the new platform; and
- transition to a full-fledged Cell ecosystem available to a steadily growing community of Cell developers via software development kit distributions. The Cell SDK is publicly available on IBM alphaWorks at www.alphaworks.ibm.com/tech/cellsw.

The Cell team turned to open source software to accelerate the development of an ecosystem for the Cell architecture.

Open source also was used to provide an environment in which to deploy proprietary tools targeted at specific high-leverage points in the Cell BE software stack, such as autoparallelizing compilers based on the IBM proprietary XL C.[4] While XL C provides a significant value proposition beyond open source tool suites, it integrates with open source assemblers, linkers, debuggers, and libraries in a seamless mixed environment.

Adopting open source allowed us to reduce the development cycle by leveraging a wide developer base with open source tool skills, leveraging tools designed for portability across platforms and providing early prototyping ability. During the exploratory phase, development occurred independent of the open source community at large, and we were able to make decisions based solely on the technology needs of the emerging architecture. Later, public distributions reflected changes made as part of the open source community adoption process and involved compromises to accommodate the cross-platform nature of the open source projects.

Open source tools were deployed in a proprietary execution environment, based on execution-driven simulators for the SPU and a Cell BE full-system simulator based on Mambo.[5]

ANATOMY OF A CELL APPLICATION

A Cell application executes in a heterogeneous architecture consisting of PPE and SPE cores, respectively implementing the Power Architecture and Synergistic Processor Architecture. To match this mix of processor elements, a Cell application consists of two classes of instruction streams corresponding to the different architectures.

In the current software architecture model, each Cell application consists of a process that can have associated PPE and SPE threads that are dispatched to the corresponding processors. When an application starts, the operating system initiates a single PPE thread, and control resides in the PPE. The PPE thread can then create further application threads executing on both the PPE and SPEs, supported by a thread management library based on the pthreads model.

SPE thread management includes additional functions, such as moving a Cell application's SPE component into an SPE's local store, transferring application data to and from the local store, and initiating execution of a transferred executable at a specified start address as part of thread creation.

Once an application has initiated the SPE threads, execution can proceed independently and in parallel on PPE and SPE cores. While the PPE accesses memory directly using load and store instructions, application components executing on the SPE use the MFC to perform data transfers to the SPE local store before accessing application data with load and store instructions. The MFC is accessible from the PPE via a memory-mapped I/O interface and from the SPU via a channel interface.

The CBEA allows a variety of programming models, including an accelerator model based on a remote procedure call, function pipelines, and autonomous SPE execution. The simplest use of the SPE is the accelerator model where the PPE transfers the working set as part of the invocation and offloads a compute-intensive function onto one or more SPEs. Developers can also compose function pipelines where each SPE performs a set of functions on a data stream and then copies its output to the next pipeline stage implemented on another SPE via the MFC. Autonomous SPE execution occurs when the application starts an SPE thread, and the thread uses its MFC to independently transfer its input data set to the local storage and copy result data to the system memory.

In these programming models, the PPE typically uses its cache-based memory hierarchy to execute several control

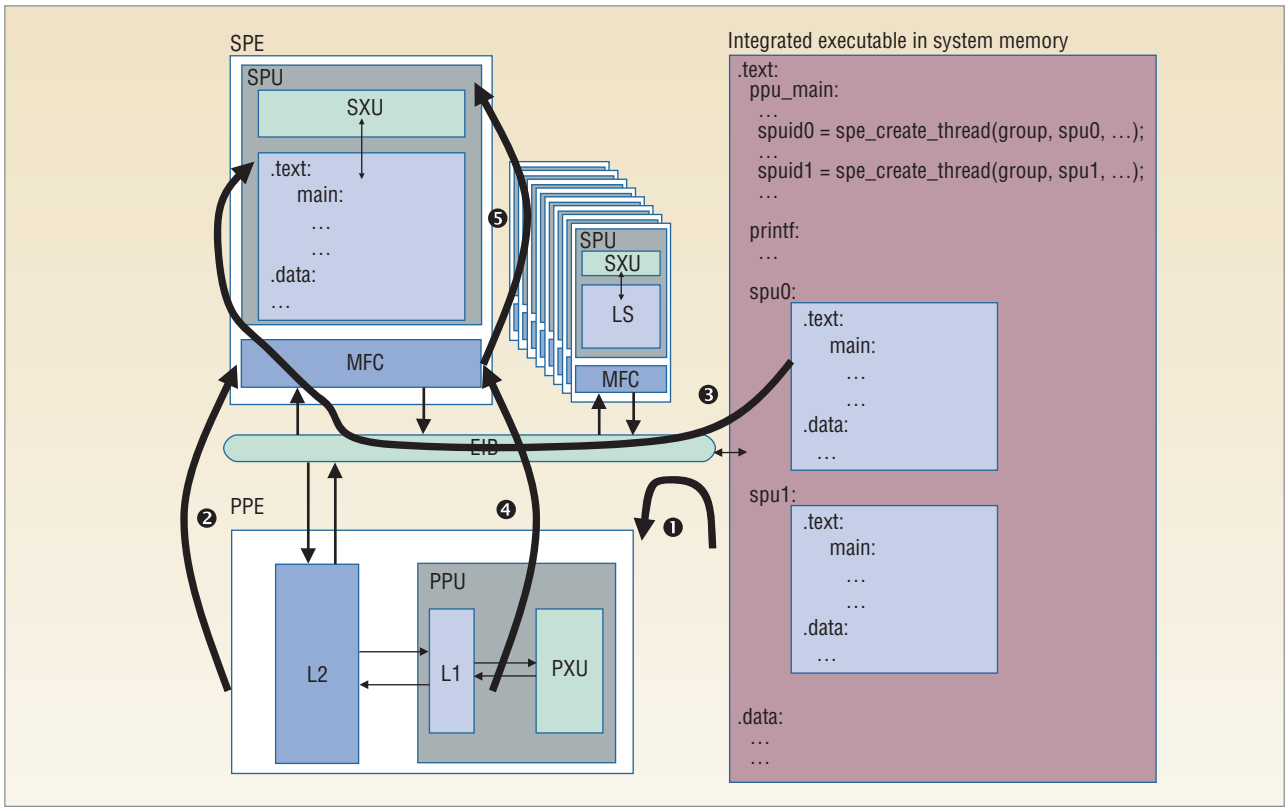

Figure 1. Execution start of an integrated Cell Broadband Engine application. (1) Power Architecture image loads and executes; (2) PPE thread initiates MFC transfer; (3) MFC data transfer occurs; (4) PPE instructs MFC to initiate SPU execution at specified address; and (5) MFC starts SPU execution.

functions, such as workload dispatch to multiple SPE data-processing threads, load balancing and partitioning functions, and a range of control-dominated application code.

Data-intensive processing

The SPE programming model is particularly optimized for the processing of data-processing-intensive applications, where the application transfers a block of data to the SPE local store and the SPU operates upon it. Computation results are stored back to the local store and eventually transferred back to system memory or directly to an I/O device by the MFC.

This processing model using SPEs to perform data-intensive regular operations is particularly well suited for media processing and numerically intensive data processing.[6] Both the SPE and PPE offer data-parallel SIMD compute capabilities to further increase the processing performance of data-processing-intensive applications. While these facilities increase the data processing throughput potential of each processor element, the key is exploiting the 10 execution thread contexts on each Cell BE chip (two PPE threads and eight SPEs).

Data multibuffering

To hide the memory access latency to the slow external memory, data transfers are best performed using data multibuffering (double buffering or even triple buffering). With double buffering, software pipelining is performed at the memory transfer level: The SPU operates on one data set in one data buffer, while the MFC transfers the next data set into another data buffer. Data multibuffering maps onto and exploits the compute-transfer parallelism in each SPE with its independent SPU execution and MFC data transfer threads.[7]

Application loading

Figure 1 illustrates application execution on the heterogeneous cores in the Cell BE. Initially, the image resides in external storage. The executable is stored in an object file format such as extensible linking format (ELF), consisting of text (read-only) and data (read/write) sections. In addition to instructions and read-only data, the text section also contains copies of one or more SPE execution images specifying the operation of one or more SPE threads.

To start the application, the operating system loads the Power Architecture object file, and (1) execution of the Power Architecture program thread begins. The application then initiates execution of application threads on the SPEs. To accomplish this, the application PPE must first transfer a thread execution image to an SPE's local store. (2) The PPE initiates a transfer of a

thread execution image by programming the MFC to perform a system memory-to-local storage block transfer, which is queued in the MFC command queues. (3) The MFC schedules the MFC request and performs a coherent data transfer.

The PPE can repeat these steps to transfer multiple additional memory-image segments containing either SPE application code, SPE libraries shared between threads, or SPE application data. When it has transferred the image, (4) the PPE issues an MFC request to start SPU execution. (5) The SPU starts execution at a specified address.

In addition to integrated executables consisting of PPE and SPE threads, Cell also can execute traditional unmodified Power Architecture executables for compatibility with industry-standard Power Architecture processors, as well as a new class of Synergistic Processor executables called *spulets*. A spulet is a Synergistic Processor Element-only program executing in a protected virtualized environment provided by the Power Architecture protection and translation model.

Using a quadword-based memory interface simplifies the data alignment logic and reduces operation latency.

COMPILING FOR A PERVASIVELY DATA-PARALLEL ARCHITECTURE

The first tool to provide any proof-of-concept prototyping capability for Cell systems, in particular the novel SPU architecture, was an execution-driven ISA simulator based on a preliminary architecture specification proposal. To simplify the development and prototyping flow, this simulator read assembly source code, and early library deployment occurred by loading multiple assembly source files.

The GNU C compiler (GCC) provided the first testing ground for the open source strategy and offered an early confirmation and proof-of-concept of many ideas introduced in the Cell BE. Before the final proposal was complete, we started development of a compiler based on GCC to demonstrate and explore the concepts introduced in the SPU—in particular, its SIMD-based architecture and the scalar layering used to implement a pervasively data-parallel computing architecture. This configuration also provided the first programming environment for library development and the first media-processing and encryption/decryption kernels that validated the newly defined architecture's performance on these critical functions.

To implement a compiler showing the feasibility of concepts the SPU architecture introduced, we leveraged the entire GCC front end, including the Power Architecture SIMD extension interface, and rewrote a back end from the ground up to support this new computing concept. This allowed us to quickly support the entire semantics of the C language, its GCC extensions, and the SIMD vector-programming intrinsic interface.

Scalar layering

One major concept of the Synergistic Processor Architecture that we needed to validate was scalar layering. Unlike prior architectures, the SPU architecture does not provide separate resources to support execution of scalar computations; instead, the compiler generates code sequences to compute scalar results with the SIMD data paths. We refer to an architecture using SIMD execution resources for scalar operations as a pervasively data-parallel computer architecture.

In the SPU architecture, all instructions take their operands from a unified 128-bit-wide vector register file with 128 architected registers. Compilers and programmers can use these instructions either to implement data-parallel SIMD operations or to produce scalar results by performing a wide result and using only the result returned in a single slot. To support scalar layering, instructions that use a single scalar input also read their operand from a 128-bit register and use the value from the "preferred slot," the vector register's first 32-bit vector element slot. This includes memory operations, which expect the memory address in the preferred slot, and branch instructions that can access a condition value or target address in the preferred slot.

In the SPU, all memory accesses operate on aligned quadwords, which must reside at addresses that are multiples of 16 bytes. To facilitate reading and writing of data values shorter than a quadword, the architecture supports efficient extract and merge operations, and memory accesses to retrieve an aligned quadword ignore the low-order four bits. Using a quadword-based memory interface simplifies the data-alignment logic and reduces operation latency. If the program is to perform access to a data value smaller than a quadword, the low-order bits indicate the data location within the quadword. The compiler expands such functionality and generates code to extract and format data explicitly using the simple SIMD RISC primitives that the architecture provides.

Although this alignment sequence requires several instructions, it reduces the overall data-flow latency because properly aligned scalar and vector data do not require alignment in most cases. For misaligned vector data, the compiler can optimize data-access patterns across loop iterations to generate more efficient alignment sequences. This new architectural concept eliminates the separate scalar execution units typically found in processors to support execution of scalar operations. Scalar layering reduces SPE area and design complexity

and increases the number of SPEs that can be placed on a same-sized chip, which improves overall system performance.

Compiler prototype

By leveraging the GCC infrastructure, we could concentrate on developing compiler support for the novel SIMD RISC architecture features rather than undertaking the lengthy and costly process of developing an entire compiler from scratch. Using this compiler, we demonstrated the feasibility of generating appropriate sequences to implement data alignment in software instead of in hardware and demonstrated that hardware complexity reduction and efficient instruction scheduling result in an overall faster implementation.

The GCC also served as a vehicle to prototype an application binary interface (ABI) by experimenting with calling conventions and stack frame layouts and prototyping a first set of support libraries. The SPU ABI adopts the preferred-slot concept for passing scalar variables as function arguments and results and for allocating scalar variables in globally allocated registers as the default location for scalar data within a register file. Advanced compilers with intraprocedural optimization capabilities can optimize placement of scalar data in any slot.

To provide a consistent language interface for programmers between the PPE and SPE code, we adopted the same language interface to vector data types for the SPE as was already provided for the PPE. Similar to the Power Architecture vector specification, the SPU programming model also uses polymorphic intrinsics where the data type specifies the intrinsic operation—much as the operator "+" specifies either integer or floating-point operation based on its operands' data type.

Seeding Cell application development

The development of the GCC-based SPU compiler proved the viability of the SPU architecture concept. Library and application developers adopted the compiler soon after it could compile the first programs and before full functionality became available. This had the desired effects of seeding a high-level-language (HLL)-based library and kernel development effort (which evolved into the SDK distribution), as well as giving valuable feedback from application developers to the Cell software and architecture teams.

By providing an early high-level development environment, the open source strategy also addressed a form of Clayton Christensen's innovator's dilemma[8] by preventing the emergence of a tuned assembly code base. Invariably, such an assembly code base would have outperformed any nascent, unoptimized HLL codes, drawing attention and efforts from the development of the HLL code, slowing or even completely forestalling development of the HLL library code. Using HLLs ultimately provides advantages in terms of programmer productivity and ease of adoption of new algorithms and data structures; thus, it delivers significant returns in performance or functionality.

Cell GCC became available in 2001, and we used it for all code development for the first two years until the XLC compiler became available. GCC-based compilers continue to be an important part of the Cell BE software ecosystem.

The first programming support specific to the Cell BE targeted the SPU to explore the new architecture.

HETEROGENEOUS ARCHITECTURE TOOLS

Supporting software development in a heterogeneous architecture represents a set of challenges surpassing traditional application build environments. Integrating tools across different architectures is key to allowing programmers to focus on application development and ensuring their productivity. To address this need for a cohesive application development and build environment, we used a multipronged approach, reflecting the options available for different tools.

The initial tool environment started out hand in hand with the architecture definition work. A small team concentrated on developing key functionality and exploring the new architecture. The first programming support specific to the Cell BE targeted the SPU to explore the new architecture. Software and hardware development occurred in parallel, and we developed the SPU specification, compiler, and simulator infrastructure in parallel as we explored different design choices.

As the architecture evolved and the developers wrote longer programs, they needed a more robust development environment. We accomplished this by porting the GNU binutils to the SPU, providing a robust assembler, linkage, and binary manipulation utilities.

At the same time, integration between PPE and SPE to support advanced application development became more pressing. Ideally, this environment would provide a single, common interface for PPE and SPE program build with the ability to specify the target processor element on the command line. In a next step, a compiler would then automatically build Cell applications, partition the program into functions to be executed on the PPE and SPEs, respectively, and insert thread synchronization and data transfer as necessary for the correct execution of the program.

Integrated compilation

We defined the compiler to share a common vector programming model and support migration of applica-

tion source code between the different processor element types. Based on the common type system to represent vector data, we provided low-level intrinsics to access the specific architecture features of the two processor elements.

To compile an application for a Cell BE processor, portions of the program must be compiled specifically for each processor type. To accomplish this, compilers are provided for both processor element targets with separate executables for PPE and SPE, which are built from common source code. This makes traditional compiler optimizations and newly developed SIMD vectorization support available for both processor elements. To provide a common compilation interface for PPE and SPE, the compiler driver can invoke the proper executable for each target type based on a specified target architecture.

We experimented with several generations of SPE enablement in Linux to derive the most efficient and programmer-friendly model.

Building integrated executables

The GNU binutils provide a highly portable binary utilities tool chain with architecture versioning support. Thus, we chose to provide assembler and linker support for both PPE and SPE targets with a single binary. The linker generates object files in ELF format for both PPE and SPE. Finally, as Figure 1 shows, we developed an embedder program to build an integrated executable by including SPE executables in PPE executables, such that a thread executing on a PPE can initiate a thread executing the code the SPE binary specifies.

The embedder reads one or more fully compiled and linked SPE ELF binaries and embeds the SPE program in the integrated Cell executable in ELF format. The resulting PPE executable contains the PPE code, multiple embedded SPE executables, and management functions for transferring the SPE code to an SPE.

To embed an SPE executable in a PPE program, the embedder reads the fully linked SPE executable, extracts the memory image (both instruction and data), and generates C code containing data arrays corresponding to the memory image (data and text segment). It then invokes the PPE compiler to generate an object file with the data array holding the executable, which can be linked to PPE object files to give a single Power Architecture executable containing SPU object modules.

USING LINUX IN HETEROGENEOUS ARCHITECTURES

The Linux operating system played a central role in the STI development process. We based the initial port to the Cell BE on the Linux 2.4 kernel's 64-bit Power Architecture distribution and bootstrapped it on the Mambo full system simulator long before the design was finished. A key advantage of this approach was that it allowed exploration of heterogeneous execution models and evaluation of software support for proposed architecture functionality.

Porting Linux to the Cell BE involved addressing two important challenges. From a programming model perspective, we had to explore programming paradigms to enable applications to efficiently use the SPEs; from an operating system design perspective, the engineering challenge revolved around the dramatic break with the kernel's expectations—namely, that each processor would be handling its own memory-mapping needs. While centralizing system management functions (such as virtual memory management) is one of the enablers of Cell's efficiency, special consideration must be given to this aspect in porting legacy operating systems.

We experimented with several generations of SPE enablement in Linux to derive the most efficient and programmer-friendly model. From a programmability perspective, a key challenge was making SPEs easily accessible without imposing numerous constraints that would complicate application development. As we addressed these issues, we provided several experimental prototypes to early adopters to gather feedback. Based on real-world programming requirements and feedback from those developers, we evolved a generic and flexible SPE thread model. We based this model on the familiar pthreads concepts using the Linux 2.6.3 kernel source base and providing a heterogeneous lightweight thread model where a system call could spawn an SPU process, as Figure 2 shows.

Fault handling

From an operating system design perspective, a key challenge was to handle exceptions delivered on behalf of SPEs. This was a novel architectural mechanism, which had not been planned for in the internal Linux architecture. This model broke with traditional operating system kernels in one significant way: In normal symmetric multiprocessor system kernels, exceptions are associated with the currently scheduled process and can deliver only a single exception to the operating system at a time. In contrast, a Cell system could simultaneously deliver eight SPE exceptions to a single PPE, which also must handle its own PPE-related exceptions.

To address page-fault handling, we adopted an innovative deferred SPE exception approach in which the exception handler collects and preserves the relevant SPE fault information. A new deferred SPE page-fault handler then uses this information, executing in a kernel thread and implementing a Power Architecture-compliant page-fault handling routine—acquiring

spinlocks, sleeping, and so forth, as needed. Because the kernel thread executes the page-fault code at noninterrupt priority, it can spin on locks or sleep while waiting on a page transfer from external storage without causing deadlocks that might be introduced if multiple page-fault handlers were active simultaneously.

Thread management

To support a flexible SPE programming environment and provide a familiar programming abstraction, we created an SPE thread management API similar to the Posix pthreads library. This API supports both the creation and termination of SPE tasks and atomic update primitives for ensuring mutual exclusion. The API can access SPEs using a virtualized model wherein the OS dynamically assigns SPE threads to the first available SPE. This API completely virtualizes SPEs and the number of SPEs provided in a specific CBEA implementation or hypervisor-created partition. Optionally, applications can use a program-specified affinity mask to assign SPE threads to specific SPEs.

Interelement thread communication and synchronization architecture features (mailboxes, signal delivery, and so on) can be accessed either through a set of system calls or by allowing the user application to map an SPE's memory-mapped control block into its application space. In the CBEA, the SPE control block actually consists of three separate control blocks corresponding to functions to be accessed by a user space application, an operating system, and a hypervisor. Using the user-accessible function control block, an application can perform direct MMIO operations between processor elements to communicate between SPEs and remote elements (either SPEs or the PPE) and avoid the overhead associated with system calls.

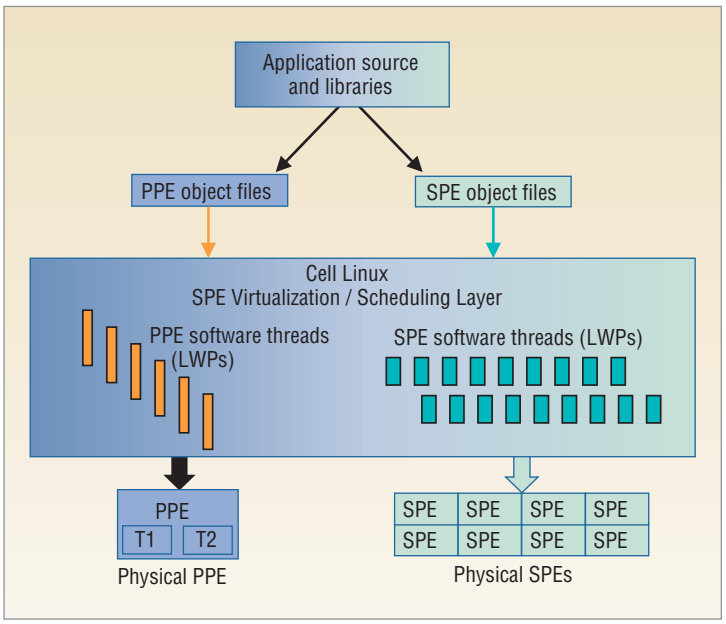

Figure 2. Application development and execution for a heterogeneous chip multiprocessor such as the Cell BE. An application program and libraries are partitioned into a set of functions executing on the PPE and SPE and compiled into object files for the PPE and SPE, respectively. The object files are then linked into an integrated executable (shown in Figure 1). The PPE object files contain code for several PPE software threads, and the SPE files contain code for several SPE software threads. When the application executes on a Cell-aware operating system (such as Cell Linux), it creates software threads using the thread library and the operating system services providing software threads ("lightweight processes" or LWPs) for the PPE and SPE. The operating system then maps the software threads to the available hardware threads in a Cell system. In the first implementation, each Cell BE chip offers two PPE hardware threads using hardware multithreading in the PPE core and eight single-threaded SPEs.

When the application requests creation of a thread, the SPE thread library requests the OS to allocate an SPE and creates SPE threads from SPE ELF object format files wrapped into an integrated Cell executable. To offload a portion of thread initialization onto the SPE, the PPE can use a "miniloader" executing on the SPE to perform SPE program loading. The miniloader, a 256-bit SPE program, downloads the application ELF segments from the host thread's effective address space to the SPU local store. Using an SPE-side miniloader is advantageous because it offloads the PPE from having to pace program loads and it can use the SPE miniloader to preinitialize registers with application/OS parameter values.

This is attractive because multiple SPEs can load threads simultaneously, and SPEs have deeper fetch queues to hold multiple block transfer requests associated with loading a thread. In addition, communication within a processor element's scope—that is, between the SPU and its associated MFC—is more efficient than interprocessor element communication between the MFC in an SPE and the PPE using MMIO.

Debugging integrated executables

The Cell BE requires an advanced debugging environment to allow developers to track applications executing on up to nine cores in a heterogeneous environment. Application developers working on a Cell BE application need to be able to follow the flow of control from one processor element to another processor element, from the PPE to a task spawned on the SPE, or from one SPE to the next.

The Cell debugging environment is built on the GNU debugger (GDB) and is the Cell debugging solution for both the GCC open source compiler and the IBM proprietary XL C compiler. The Cell debugging environment, however, goes far beyond a simple port of the

GDB debugging tool. To take advantage of the Cell BE's unique characteristics, the environment exploits additional system services to offer application debugging in a heterogeneous multicore architecture. When a Cell BE application spawns an SPE thread, GDB will follow that newly created SPE thread with the ability to properly interpret executables for the SPU architecture.

As both PPE and SPE debuggers are based on the common GDB source, PPE and SPE debuggers offer a consistent user interface. Initially, starting a thread instantiated a new processor-element-specific instance of the debugger; more recent versions support PPE and SPE debugging with a single heterogeneous debugger. Unless the developer selects an assembly language view of the program, the source-level debugger makes Cell's heterogeneous architecture completely transparent, allowing the developer to concentrate on the application behavior without regard to underlying instruction set architecture.

The Cell multicore debugging environment is based on several components:

- a GUI tracking multiple threads on the PPE and SPEs (an alternative text-based debugging environment is also available);
- GDB as the debugger engine, allowing developers to follow the execution of code across the PPE and SPEs, set breakpoints, and display data values stored in registers and memory; and
- debugging support in the system software stack that allows GDB sessions to gain control of a thread when it is initiated as well as interfaces to implement state inspection and modification.

The heterogeneous debugger architecture depends on support in the ABI—specifically, the thread creation interfaces provided in libspe.a, the SPU support library. Thus, all applications built with the standard Cell BE libraries automatically benefit from transparent heterogeneous debug support. To accomplish this, libspe.a and the dynamic library loader (ld.so) include support (during SPE thread creation) to allow ppu-gdb to obtain control at predictable points and retrieve information necessary to debug code in a newly created SPE thread. We have also included support for the debug environment in the SPU linker (spu-ld) by generating context information. This allows the debugger to find the symbol tables and other debugging information for each SPE thread when an application developer initiates an spu-gdb session.

The architecture, operating systems, and Cell system ABIs tightly integrate heterogeneous debug support. As an example, programmers can set arbitrary breakpoints in an SPU program at the source level. The GDB then translates this breakpoint into a location in the SPU local store and inserts an SPU "stopd" instruction. When the SPU attempts to execute this instruction, the SPE delivers an interrupt to the PPE. In response to this interrupt, the kernel will perform a context save of the SPU thread state and send a SIGSTOP signal to the tracing process, allowing the debugger to take control when the application reaches a breakpoint.

The SPU GDB supports access to both the program state of user programs in the SPU and access to SPE state to provide a comprehensive view of application execution in a Cell system. In addition to SPU application state, this includes other SPE state corresponding to program-initiated operations such as mailbox communications, DMA transfers maintained in the MFC, and so forth.

We used open source software across the entire system stack to explore novel architecture concepts and their software enablement. We architected the software stack to present a high-level language programming environment abstracting specific architecture choices. The software environment allows application developers to focus on exploiting application parallelism to deliver the superior Cell performance as actual application performance. Using open source software has allowed accelerating architecture validation and debugging in a full-fledged software environment. In addition to being highly useful during the later stages of architecture definition and refinement, this approach also has provided an environment for early Cell adopters.

We have benefited—in real-world applications and in real time—from the feedback of Cell adopters in exploring programming abstractions for an integrated heterogeneous environment as pioneered by the Cell Broadband Engine Architecture. Many of the tools that formed the basis of the Cell BE infrastructure are still in use today, while others have served as a testbed and will coexist with commercial tools in a rich Cell software ecosystem. Adopting an open software strategy has allowed us to accelerate the market deployment of a new architecture offering innovations to improve efficiency and performance across the entire architecture stack by prototyping innovative software solutions while building on a familiar environment.

Finally, the Cell BE software environment allows application programmers to deliver high performance by focusing on applications, not the architecture or an unfamiliar tools environment. The true success of the Cell software environment is to allow the development of new, previously unseen applications for the Cell BE.

Acknowledgments

The authors thank Jim Kahle, Mike Day, and Ted Maeurer for their leadership and support. They also

thank Peter Hofstee, Ted Maeurer, Dan Prener, Valentina Salapura, and John-David Wellman for their many insightful comments and suggestions during the preparation of this article.

References

1. J. Kahle et al., "Introduction to the Cell Multiprocessor," *IBM J. Research and Development*, Sept. 2005, pp. 589-604.
2. M. Gschwind et al., "Synergistic Processing in Cell's Multicore Architecture," *IEEE Micro*, Mar./Apr. 2006, pp. 10-24.
3. M. Kistler et al., "Cell Multiprocessor Communication Network: Built for Speed," *IEEE Micro*, May/June 2006, pp. 10-23.
4. A. Eichenberger et al., "Optimizing Compiler for the Cell Processor," *Proc. 14th Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 2005), IEEE CS Press, 2005, pp. 161-172.
5. P. Bohrer et al., "Mambo—A Full System Simulator for the PowerPC Architecture," *ACM Sigmetrics Performance Evaluation Rev.*, Mar. 2004, pp. 8-12.
6. S. Williams et al., "The Potential of the Cell Processor for Scientific Computing," *Proc. ACM Computing Frontiers 2006*, ACM Press, May 2006, pp. 9-20.
7. M. Gschwind, "Chip Multiprocessing and the Cell Broadband Engine," *Proc. ACM Computing Frontiers 2006*, ACM Press, May 2006, pp. 1-8.
8. C. Christensen, *The Innovator's Dilemma*, McGraw-Hill, 1997.

Michael Gschwind *is an architect and design lead for a future server system at IBM T.J. Watson Research Center, where he helped develop the Cell Broadband Engine Architecture concept and was a lead architect in defining the Synergistic Processor Architecture. He developed the first Cell BE compiler and helped initiate and contributed to the creation of the Cell software environment. His research interests include power-efficient high-performance computer architecture and compilation techniques. Gschwind received a PhD in computer science from Technische Universität Wien. He is an IBM Master Inventor and a senior member of the IEEE. Contact him at mkg@us.ibm.com.*

David Erb *is a developer currently working on Cell Ecosystem Development at IBM Austin. His research interests include performance tuning for multicore and SIMD systems. Erb received an MS in electrical engineering from the University of Texas at Austin. Contact him at djerb@us.ibm.com.*

Sid Manning *is an advisory software engineer working on Cell solution architectures and development at IBM Austin. His current interests are development tools for multicore architectures. He received a BS in computer science from the Rochester Institute of Technology. Contact him at sid@us.ibm.com.*

Mark Nutter *is a senior software engineer and IBM Master Inventor working for IBM's Systems & Technology Group. His responsibilities include adapting algorithms, programming models, and OS software to multicore processors such as the Cell Broadband Engine. His research interests include the development of an interactive ray tracer for the Cell BE and large model visualization. Nutter received a BS in computer science from the University of New Mexico. Contact him at mnutter@us.ibm.com.*