Introduction to the Cell Broadband Engine Architecture

C. R. Johns
D. A. Brokenshire

This paper provides an overview of the Cell Broadband Engine™ Architecture (CBEA). The CBEA defines a revolutionary extension to a more conventional processor organization and serves as the basis for the development of microprocessors targeted at the computer entertainment, multimedia, and real-time market segments. In this paper, the organization of the architecture is described, as well as the instruction set, commands, and facilities defined in the architecture. In many cases, the motivation for these facilities is explained and examples are provided to illustrate their intended use. In addition, this paper introduces the Software Development Kit and the software standards for a CBEA-compliant processor.

Overview

The Cell Broadband Engine† Architecture (CBEA) defines a family of heterogeneous microprocessors that target multimedia and compute-intensive applications [1]. The CBEA resulted from a joint effort among the Sony Group, Toshiba, and IBM to develop the next-generation processor. The following motivations shaped the development of the architecture [2]:

- Provide outstanding performance on computer entertainment and multimedia applications.
- Develop an architecture applicable to a wide range of platforms.
- Enable real-time response to the user and the network.
- Address the three design challenges facing traditional processors: memory latency, power, and frequency.

The CBEA extends the IBM PowerPC* 64-bit architecture with loosely coupled, cooperative offload processors. The CBEA is set apart from other processor architectures by the use of two independent instruction sets: the PowerPC and the synergistic processor unit (SPU) instruction sets. For a processor to be considered CBEA compliant, the processor must contain one or more PowerPC processor elements (PPEs), one or more synergistic processor elements (SPEs), and the required feature set defined by the CBEA. **Figure 1** is a block diagram of a CBEA processor.

The PPE is compliant with the IBM PowerPC Architecture* [3]. It is intended to perform the system management and application control functions, or "control plane" processing. Whereas other architectures have augmented the instruction set of the processor with tightly coupled extensions such as the vector/single-instruction multiple-data (SIMD) multimedia extension, the CBEA employs an independent SPU that is compliant with the SPU instruction set architecture [4] to perform the compute-intensive, or "data plane," processing. This allows the data processing and control functions to be decoupled, enabling more application parallelism.

Another distinction between a traditional processor and the CBEA is the definition of two storage domains: main and local. The main storage domain is the same as that commonly found in most processors. This domain contains the address space for system memory and memory-mapped I/O (MMIO) registers and devices. Associated with each SPU is a local storage address space, or domain, containing instructions and data for that SPU. Each local storage domain is also assigned an address range in the main storage domain called the *local*

0018-8646/07/$5.00 © 2007 IBM

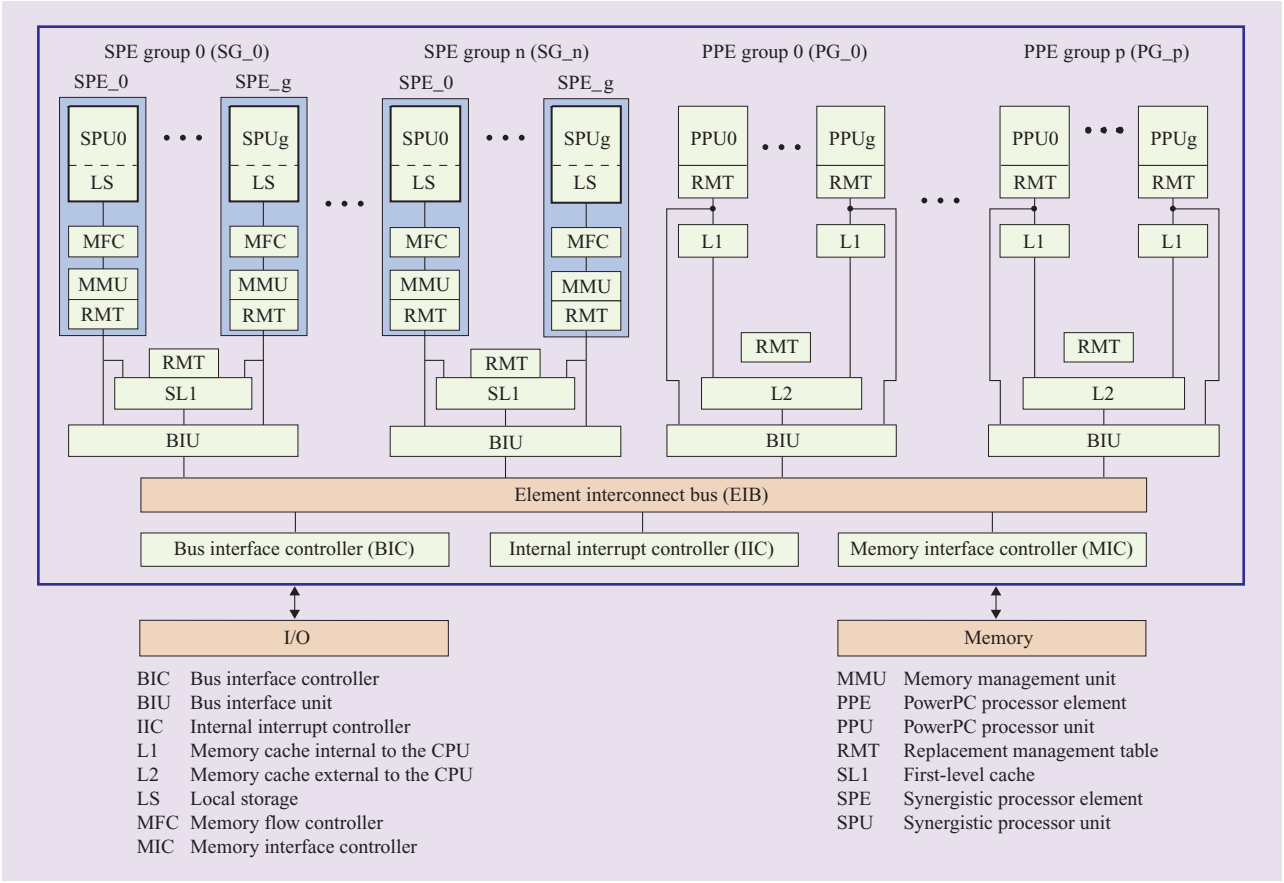

Figure 1

CBEA processor block diagram.

storage alias. The SPU can address memory directly only within the associated local storage. To access data in the main storage domain and maintain synchronization with other processors, each SPU and local storage pair is coupled with a memory flow controller (MFC). The combined SPU, associated local storage, and MFC make up the SPE. In addition to the PPE and the SPE, the CBEA includes many features for real-time applications not typically found in conventional processor architectures.

The "Cell Broadband Engine Architecture" document, which defines the architecture, refers to the "PowerPC Architecture Book" and the "SPU Instruction Set Architecture" document to avoid duplicating information. For a complete definition of the CBEA, the reader must have access to all of these documents, which are publicly available on IBM Web pages [1, 3, 4].

The CBEA document is divided into two parts: the user-mode environment (UME) and the privileged-mode environment (PME). The UME defines the instructions and facilities required for application portability. This section is similar to Books I and II of the PowerPC Architecture. For the PowerPC Architecture, only Book I compliance is necessary for application portability. Since the CBEA defines a heterogeneous multiprocessor, the synchronization features found in Book II of the PowerPC are necessary for cooperative processing. For this reason, Books I and II were collapsed into a single UME section. The PME, which defines the instructions and facilities required for operating system and hypervisor[1] development, is similar to Book III of the PowerPC Architecture.

PowerPC processor element

Within a CBEA-compliant processor, the PPE performs the control plane functions that typically required the more general-purpose computing provided by the PPE. The PPE is based on Version 2.02 of the PowerPC Architecture, which offers many of the features required for the application spaces targeted by the CBEA. The

[1]A *hypervisor* is a layer of software running on the processor that allows multiple "guest" operating systems to run concurrently. The hypervisor virtualizes the processor and the system resources.

PPE is based on the PowerPC Architecture for four reasons: 1) The PowerPC Architecture is a mature architecture that is applicable to a wide variety of platforms. 2) It supports multiple simultaneous operating environments through logical partitioning. 3) It contains proven microarchitectures that meet the frequency and power challenges of the targeted market segment. 4) Use of the PowerPC Architecture leverages the IBM investment in the PowerPC ecosystem.

A key advantage of the PowerPC Architecture is its unique ability to support multiple concurrent operating environments through the use of logical partitioning. This feature is critical to executing a non–real-time operating system (RTOS) for the user interface while simultaneously executing a non-RTOS, such as the Linux** operating system, for management of the system. Concurrent execution of an RTOS and the Linux operating system allows non–real-time processes to be performed in the background without affecting the performance of an application running under the RTOS.

Although the PowerPC is a mature architecture for more traditional platforms, concurrently supporting both an RTOS and a non-RTOS within the heterogeneous multiprocessor presented many architectural challenges: To meet these challenges and optimize the architecture for media-rich applications, the PowerPC Architecture required enhancements to the vector/SIMD multimedia extension, as well as introduction of the mediated external exception, the multiple concurrent large-page support, software management of translation lookaside buffers (TLBs), and the cache replacement management extensions.

The following subsections describe the mediated external exception extension, the vector/SIMD multimedia extension, and the multiple concurrent large-page extension. Software management of TLBs and cache replacement management are described in separate sections because they apply to both the PPEs and the SPEs. See the CBEA document [1] for more information on each extension.

Mediated external exception extension

A key attribute of an RTOS is a guaranteed interrupt latency. A single RTOS can control the interrupt latency by not allowing interrupts to be disabled for more than a predetermined amount of time. However, when multiple operating systems are running, the presentation of an interrupt can be delayed by another partition for an arbitrary amount of time. The mediated external exception extension to the PowerPC Architecture overcomes this deficiency by allowing an interrupt to be presented to the processor even if interrupts are disabled.

In the PowerPC Architecture, the term *exception* describes the interrupt condition, and the term *interrupt* refers to the processor acting on the exception condition or jumping to the interrupt handler. The critical exception conditions for an RTOS are those generated by SPEs and external devices. The current PowerPC Architecture allows an external exception to invoke a hypervisor-privileged interrupt handler. However, an operating system still has control of the external exception enabled in the machine state register of the processor. The CBEA-mediated external exception extension enables the interrupt handler to be invoked as long as the processor is not operating in hypervisor state or if external exceptions are enabled. This prevents an operating system from delaying the presentation of an exception to the processor and allows a hypervisor-privileged interrupt handler to be invoked. If the exception is for the RTOS, the hypervisor can immediately pass control to the interrupt handler of the RTOS regardless of which partition is currently active.

However, an interrupt can occur when the RTOS is in a critical section with external exceptions disabled, and this is where the mediated portion of the extension comes into play. When external exceptions are disabled by the partition that is expected to handle the interrupt, the hypervisor-privileged interrupt handler sets a mediated external exception request bit in the logical partitioning control register and returns from the interrupt. When the partition enables external exceptions, the hypervisor-privileged interrupt handler is once again invoked with a mediated external exception. The interrupt handler then sets the state of the processor to mimic the original external exception.

Vector/SIMD multimedia extension

Although the CBEA contains offload processors for vector and streaming media processing, the architecture team included the vector/SIMD multimedia extension for the PPE. This multimedia extension was included to run software developed for the vector/SIMD multimedia extension, to make it easier to develop and port applications to the SPE, and to allow applications to be parallelized across the PPEs and SPEs.

The vector/SIMD multimedia extension defined by the CBEA is very similar to the PowerPC 970* implementation. The major difference between the two is the rounding mode support. For compatibility with SPU applications, the vector/SIMD multimedia extension unit in the PPE supports the rounding modes defined by the SPU instruction set architecture.

Multiple concurrent large-page extension

The current PowerPC Architecture supports the base 4-KB page plus one additional large page to be used concurrently. The large-page size is implementation dependent. In the CBEA, many types of data structures are located in main storage, e.g., MMIO registers for the SPEs, local storage aliases, streaming data, and video

buffers. The limitation of only one large-page size places a burden on the TLBs. If a large-page size of 64 KB is selected, the number of translations needed for MMIO registers and local storage aliases is lower than that for the base 4-KB page size. However, more translations are required for the relatively large streaming buffers and video buffers in main memory. In contrast, a large-page size of 1 MB or 16 MB reduces the number of translations required for the streaming and video buffers, but it is too large for mapping the MMIO registers and local storage aliases.

To improve the efficiency of the TLBs, the CBEA augments the PowerPC Architecture by providing support for multiple concurrent large-page sizes. The memory management units (MMUs) in the SPE also support the multiple concurrent large-pages extension.

Synergistic processor element

The SPE is the cooperative offload processor in the CBEA intended for the computational, or data plane, processing functions. The SPE consists of three tightly coupled units: the SPU, the associated local storage, and the MFC. The SPU is a SIMD processor with an instruction set architecture optimized for compute-intensive and media applications: It operates only on instructions and data in the associated local storage. Decoupling the SPU from other aspects of the system provides a very deterministic processing environment for the programmer.

The SPU and associated local storage are coupled to the main storage domain and other processors by the MFC. The MFC enables software to move data between the storage domains and to synchronize with other processors in the system. Data movement and synchronization are initiated by using MFC commands. Either the SPU or another processor in the system, such as the PPE, can issue these commands. A direct memory access (DMA) controller in the MFC unit performs the data movement. All main storage accesses performed by the DMA controller adhere to the PowerPC Architecture for address translation and protection. They are performed asynchronously with respect to the SPU and all other units in the system.

Synergistic processor unit

The SPU provides the programmer with 128 registers, each of which is a 128-bit SIMD register. The large number of architected registers facilitates efficient instruction scheduling and also enables important optimization techniques such as loop unrolling. All SPU instructions are inherently SIMD operations that process data in one of four granules: sixteen 8-bit integers, eight 16-bit integers, four 32-bit integers or single-precision floating-point numbers, or two 64-bit double-precision floating-point numbers. The SIMD registers in the SPU are unified and can be an operand of either an integer or a floating-point instruction, unlike the split set of registers in the PowerPC Architecture [5].

To obtain the best performance from an SPU, the data structures of the program should be defined around the SIMD data flow. In addition, techniques such as double buffering should be employed to overlap the computation with the data movement. This technique insulates the SPU application from the latency of the system memory accesses. More programming tips can be found in [6]. Although the SPU is optimized for SIMD, scalar operations can be performed; they use the preferred slot (the upper word, or 32 bits, for 32-bit scalars) of the SIMD register.

Channel interface

As mentioned earlier, the SPU is decoupled from the system and the MFC provides the linkage to the main storage domain and other processors. A channel interface supplies the communication path between the SPU and the MFC. The CBEA defines multiple unique channels for issuing MFC commands and accessing MFC facilities. These channels are accessed by using the SPU channel instructions.

The SPU instruction set architecture defines the channels as four words wide, but the CBEA uses only one word, which is the preferred slot of the SIMD data flow. Each channel has the following set of attributes defined by the CBEA: direction, capacity, and the ability to be blocking or nonblocking.

Channels are unidirectional, with the direction defined by the CBEA. Channels used to retrieve information from or transfer information to the MFC are accessed, respectively, by using an SPU read channel (*rdch*) or write channel (*wrch*) instruction. Accessing a read channel by using a *wrch* instruction or a write channel by using an *rdch* instruction is not allowed and results in an invalid channel error.

Channel capacity (depth) defines the number of words that can be contained within the channel. The CBEA defines the depth for some channels; for others, it is an implementation-specific parameter. A channel count, which is accessed by using an SPU read channel count (*rdchcnt*) instruction, is used to track the amount of information in the channel. The value returned by the *rdchcnt* instruction indicates the number of valid words contained in a read channel and the available free space or number of words that can be written to a write channel.

The CBEA defines each channel as either *blocking* or *nonblocking*. A nonblocking channel essentially has an infinite depth, and a value of one is always returned when reading the channel count. Blocking channels have a

depth of one or more, with an active channel count. Accessing a blocking channel whose count is zero stalls the instruction processing of the SPU until the channel count becomes nonzero. The blocking attribute relieves the programmer from having to check the channel capacity before accessing the channel. When no other useful work can be performed, this attribute allows the program to enter a very low-power state while waiting for the channel to become available. If useful work can be performed, the programmer can check the channel capacity with the *rdchcnt* instruction before accessing the channel and can then occasionally poll the channel count while performing other useful work.

An SPU interrupt offers an alternative to polling. The presence of an SPU interrupt can affect the SPU instruction sequencing in one of two ways. If interrupts are disabled, a special SPU branch instruction (*bisled*) can be executed to branch to a target address if an interrupt is present. If interrupts are enabled, the SPU executes the next instruction from address zero in local storage and disables interrupts. The address of the next instruction that would have been executed if the interrupt were not present is saved in the SPU state save and restore register (SRR0). SRR0 can also be accessed by using two channels, one to read SRR0 and one to write SRR0. The SPU *iret* instruction can be used to return to the address stored in SRR0. SPU interrupts should not be confused with interrupts generated by an SPE that are targeted for the PPE (SPE interrupts).

The SPU event facility can be programmed to detect key SPE conditions and other system events. The event facility is accessed by using channel instructions to specify the conditions of interest and determine the event status. Software can use the event facility to stall the SPU while waiting for multiple different events or to generate an interrupt when an enabled event occurs. The SPU event facility is described in more detail later in the paper.

Memory flow controller

The MFC provides the communication path from the SPU and the local storage domain to the main storage domain and other processors in the system. The MFC essentially decouples the SPU from the main storage domain; an application can view the SPU as having an additional asynchronous load–store processor. If used properly, the MFC can insulate an SPU application from the latency of system memory.

Figure 2 is a high-level block diagram of the MFC unit. The bus interface unit (BIU) provides the interface to the element interconnect bus (EIB) [7]. The CBEA does not define the EIB protocol. The first-level cache (SL1) is a caching structure for MFC accesses to the main storage domain, and it provides an architectural entity for performance enhancements. In most implementations,

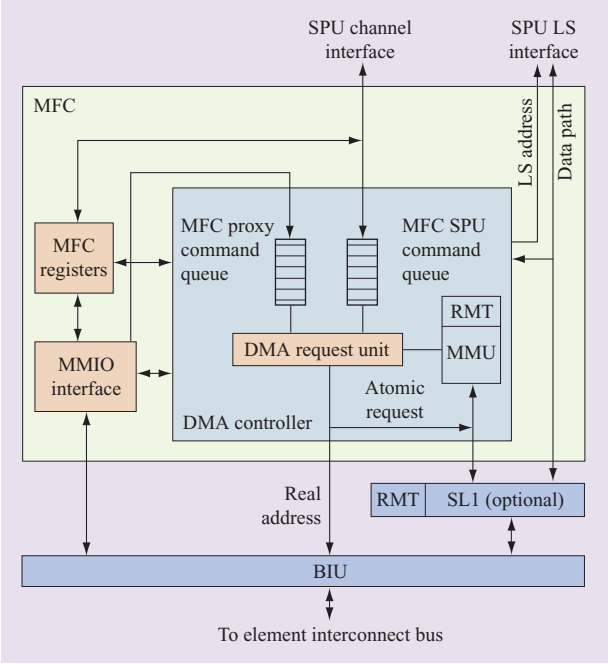

Figure 2

MFC block diagram.

the SL1 either is omitted or is very small because of the streaming nature of the MFC commands. The MFC SPU command queue is dedicated to MFC commands issued by the SPU using the channel interface. Other processors use the MFC proxy command queue to transfer data between the storage domains on behalf of the associated SPE. The MFC proxy commands, which are issued by using the MFC MMIO registers, are used primarily to efficiently initialize the local storage before an SPU program is executed. They are typically used by software executing in a PPE.

The DMA controller transfers instructions and data between SPU local storage and main storage. Programs running on the associated SPU, a PPE, or another device can issue MFC DMA commands. **Table 1** shows the MFC DMA commands supported by the DMA controller.

A single MFC DMA command can transfer up to 16 KB of sequential data between the storage domains. A DMA transfer is typically performed as a series of smaller bus transfers, usually a cache line or less. If required, coherency is maintained for these transfers as the storage attributes in the page table.

The parameters listed below affect the operation of MFC DMA commands:

- CL MFC class ID.
- TG MFC command tag identification.

Table 1 MFC DMA commands supported by the DMA controller.

Command	*Description*
get<f,b>[s]	Moves data from the effective address within main storage to local storage.
getl<f,b>[s]	Same as *get<f,b>*, except that the effective address and size for multiple transfers are specified by list elements in local storage.
put[r]<f,b>[s]	Moves data from local storage to the effective address within main storage.
	The optional "r" modifier provides a hint to the system that the data is likely to be accessed soon by the PPE. The data transferred by the command is a candidate for copying into the PPE cache.
put[r]l<f,b>	Same as *put[r]<f,b>[s]*, except that the effective address and size for multiple transfers are specified by list elements in local storage.
sdcrt	SL1 data cache range touch. Brings a range of effective addresses into the SL1 (performance hint for *get* commands). Similar to the PowerPC *dcbt* instruction.
sdcrtst	SL1 data cache range touch for store. Brings a range of effective addresses into the SL1 (performance hint for *put* commands). Similar to the PowerPC *dcbtst* instruction.
sdcrz	SL1 data cache range zero. Writes zeros to the contents of a range of effective addresses. Similar to the PowerPC *dcbz* instruction.
sdcrst	SL1 data cache range store. Stores the modified contents of a range of effective addresses. Similar to the PowerPC *dcbst* instruction.
sdcrf	SL1 data cache range flush. Stores the modified contents of a range of effective addresses and invalidates the block. Similar to the PowerPC *dcbf* instruction.
sndsig<f,b>	Sends a signal to another SPE. Updates the signal notification registers in another SPE.
barrier	Orders all commands issued prior to the barrier command with respect to all subsequent commands.
mfceieio	Orders the storage transactions caused by *get* and *put* commands. Similar to the PowerPC *eieio* instruction.
mfcsync	Orders DMA *put* and *get* operations within the specified tag group with respect to other processing units and devices on the system. Similar to the PowerPC *sync* instruction.
getllar	Get lock line and reserve–immediate. Similar in function to the PowerPC *lwarx* and *ldarx* instructions.
putllc	Put lock line conditional–immediate. Similar in function to the PowerPC *stwcx* and *stdcx* instructions.
putlluc	Put lock line unconditional–immediate. Main storage location updated regardless of reservation ownership.
putqlluc	Same *as putlluc* except queued with other DMA commands and has an implied fence modifier.

Note: The optional "s" modifier starts the SPU when the command completes.
The fence (<*f*>) and barrier (<*b*>) modifiers are used for command ordering. See the section on command ordering for the definition of these modifiers.

- TS MFC transfer size.
- LSZ MFC list size.
- LSA MFC local storage address.
- EAH MFC effective address high.
- EAL MFC effective address low.
- LA MFC list local storage address.
- LTS List element transfer size.
- LEAL List element effective address low.

Each command includes a classID parameter. The parameter contains a replacement management class identifier (RclassID) that controls the cache replacement management facility. The cache replacement management facility is described later.

Software uses a TclassID identifier to classify the type of storage being accessed. The TclassID provides a hint to the MFC and to the system about the way the storage request should be treated. For example, the TclassID can be used by the bus interface unit in the MFC to prevent accesses to a slow device from blocking accesses to the higher-bandwidth main memory. To accomplish this, the EIB transfer request queue in the MFC (not shown in Figure 2) is segmented into two or more areas. The number of entries in each segment is sized according to the latency and bandwidth characteristics of the storage being accessed. EIB transfer requests created for a command are placed in only the transfer request queue segment that corresponds to the TclassID of the command. By issuing a command with the appropriate TclassID, software can prevent the transfer request queue from filling with transfer requests to slower devices. Since the transfers to slower devices are limited to one area of the request queue, transfers accessing higher-bandwidth

devices are guaranteed a minimum number of request queue entries. This minimizes the latency impact created by accesses to slower devices. Without this feature, the transfer request queue would fill up with bus requests for slower devices and prevent the MFC from issuing requests for higher-bandwidth devices.

List commands

The CBEA provides a list modifier <*l*> for MFC DMA commands to move data that is scattered in the main storage domain. A command with a list modifier is called a *list command*. List commands are converted to a series of commands, each of which is described by a list element in local storage. Each list element in local storage contains an effective address low parameter and a transfer size parameter. When a list command is executed, the MFC reads the list elements from local storage and creates a series of DMA commands. Each DMA command that is generated has the same set of parameters as the original list command, with the exception of the transfer size and effective address low parameters. Each MFC DMA list command can contain up to 2,048 list elements, each transferring up to 16 KB of data. Only an SPU can issue list commands, and such commands cannot be issued to the proxy MFC command queue.

The list element includes a stall-and-notify flag. When set, the flag causes the MFC unit to stop executing the list command and notify the SPU after all of the DMA transfers for the list element and all previous list elements are complete. Subsequently, software can resume the list command. Among other uses, this feature allows the programmer to specify a data transfer larger than the available buffer space in local storage. The stall-and-notify flag is placed on the last element that fills the buffer. When the data transferred by the previous elements is processed and buffer space made available, the program can resume the MFC DMA list command. The CBEA defines one read channel that identifies the list commands that are stalled. It defines a write channel that is used to acknowledge the stall and allow the MFC to continue executing the list command.

Memory management unit

The MMU allows an SPU application to use the same effective address as that used by a PPE application to access main storage.

The effective address for an MFC command is provided in two parameters, the EAH parameter and the EAL parameter. The EAH parameter contains the high-order 32 bits of the 64-bit effective address, and the EAL parameter contains the low-order 32 bits of the address. The EAH parameter is optional; if it is omitted, the high-order 32 address bits are set to zero.

The MMU translates the effective address into the real address of main storage. This translation is compatible with the virtual address translation mechanism defined by the PowerPC Architecture. The PPE handles all MFC translation faults. An MFC translation fault occurs when a translation cannot be found for an effective address.

The MMU can use either the same page table as that used by the PPEs or an independent page table; it also supports the multiple concurrent large-page extension described earlier in this paper. In addition, the MMU supports software management of the TLBs, which is presented subsequently.

Command ordering

As MFC commands are issued, they are placed in the appropriate command queue. MFC commands can be executed and completed in any order, regardless of the order in which they were issued. The out-of-order execution of commands allows the MFC to use system resources efficiently to achieve the best performance. For example, if the EIB supports simultaneous reads and writes, the MFC can simultaneously execute one get and one put command. If ordering were implied by the issue order, this would not be possible.

While out-of-order command execution can help performance, in some cases, software may require strict command ordering. To achieve command ordering, the CBEA provides two command modifiers (fence <*f*> and barrier <*b*>) and a barrier command. The command modifiers order commands only within the same tag group (that is, all commands issued with the same tag parameter to the same queue). Hence, these are called *tag-specific* modifiers. A command with a fence modifier is performed after all previously issued commands within the same tag group. Commands issued after a command with a fence modifier are not affected. A command with the barrier modifier is performed after all previously issued commands within the same tag group. Commands that are issued after a command with a tag-specific barrier are also performed after previously issued commands. Some commands have an implied tag-specific barrier modifier.

When command ordering is required to be independent of the tag group, a barrier command can be issued. Regardless of the tag identifier, this command orders all previously issued commands with respect to all subsequently issued commands within the same command queue. The CBEA does not provide a mechanism to order commands with respect to commands in the other command queue.

Storage access ordering

In addition to command ordering, software can also require the EIB transfers generated by the DMA

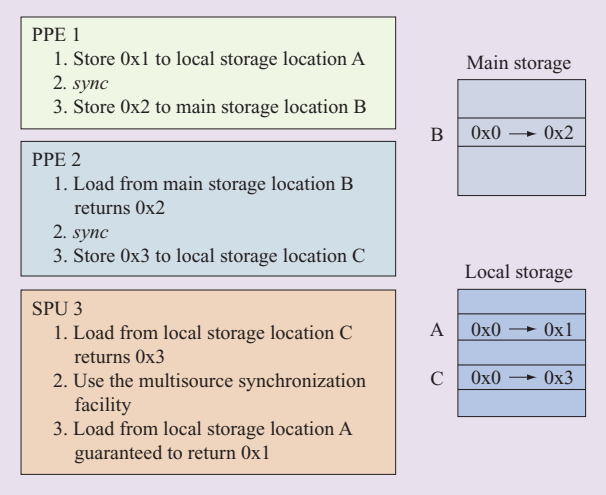

Figure 3

Example of multisource synchronization.

command to be ordered with respect to other processors and devices in the system. Although the command modifiers and barrier command provide for command ordering, they do not provide ordering of the bus transfers with respect to other processors and devices in the system. For this type of storage ordering, the *mfceieio* and *mfcsync* commands are used in combination with the command modifiers and barrier command. The *mfceieio* command provides storage ordering that is similar to the PowerPC *eieio* instruction; the *mfcsync* command provides storage ordering that is similar to the PowerPC *sync* instruction.

The ordering of storage accesses performed by two or more processors with respect to another processor or device is called *cumulative ordering*. The CBEA follows the PowerPC rules for cumulative ordering when all accesses are performed within the main storage and the proper synchronization instructions and commands are performed. Standard PowerPC rules do not apply when the storage accesses are performed within the main and local storage domains. The CBEA multisource synchronization facility addresses this case. This facility provides software with a mechanism to ensure that all accesses from the main storage domain, targeting the associated local storage domain, are complete with respect to the SPU. **Figure 3** illustrates the use of the multisource synchronization facility.

Memory-mapped I/O space and channels

The CBEA defines several SPE facilities, some of which have already been mentioned. A *facility* is a set of MMIO registers or channels that provide a specialized function. Facilities in the CBEA are accessed from the main storage domain using MMIO registers or are accessed using the channels defined by the CBEA. The facilities defined by the CBEA provide a wide variety of functions ranging from support of context save and restore to providing synchronization with other SPEs and processors in the system. The MMIO space and channels also provide access to other miscellaneous functions.

This paper does not include all of the facilities provided by the MMIO registers and channels, but it attempts to give the reader an indication of what is available. Some of the facilities provided are listed below:

- Local storage alias.
- Command issue.
- Tag-group completion facility.
- Multisource synchronization facility.
- Mailbox facility.
- Signal notification facility.
- SPU event facility and decrementer.
- Software management of TLBs.

Local storage alias facility

Within the MMIO space of the SPE is an area called the *local storage alias* that is dedicated to local storage. It provides direct access to the local storage domain of the corresponding SPE from the main storage domain. The local storage alias is used primarily to allow the direct transfer of data from the local storage of one SPE to the local storage of another SPE. For example, if an SPE issues an MFC get command with an effective address that maps to the local storage alias area of another SPE, the data is transferred directly from the local storage of the target SPE to the local storage of the issuing SPE. Without an alias of the local storage in the main storage domain, software would be forced to copy data through system memory.

Command issue facility

The memory flow controller section of this paper describes the MFC DMA commands. Commands are issued from other processors in the system using a sequence of MMIO register stores and loads. Commands issued using MMIO registers are called *MFC proxy commands*. Code running on an SPU issues commands using a sequence of channel writes; these commands are called *MFC SPU commands*.

MFC proxy commands are typically issued by the PPE to initialize local storage efficiently before the SPU is started. To issue an MFC proxy command, software writes each parameter to the corresponding MMIO register. Some parameters can be omitted, and the last parameter written is the MFC command opcode. After writing this parameter, software reads from the MFC

command status register to determine whether the command was successfully placed in the MFC proxy command queue. The MFC command opcode register and the MFC command status register are mapped to the same address. Therefore, software need not issue a PowerPC *eieio* instruction between the last store of the opcode and the read of the command status, since the PowerPC Architecture requires that two accesses to the same location be performed in program order. Queuing a command can fail because of a command sequence error or because of insufficient room in the MFC proxy command queue. A command sequence error is typically caused when the command issue sequence is interrupted and the next process or interrupt handler issues an MFC proxy command. The command sequence error response eliminates the need to require a lock for the MFC proxy queue when issuing a command. The error due to insufficient room occurs when there is no available space in the MFC proxy command queue. In most cases, software can avoid this error by reading the MFC command queue status register to ensure that there is available space in the queue before issuing the command. If either error is reported, software must repeat the command issue sequence.

The MFC SPU command sequence is a series of channel write instructions. The last channel written in the sequence is the MFC command opcode channel. This channel is blocking and has a depth equal to the size of the MFC SPU command queue. If the queue is full, the channel write results in an SPU stall until space in the queue is available. The stall is a very efficient mechanism for power savings when useful work cannot be performed until after the command has been queued. Software can avoid the stall by using the procedure described in the channel interface section or by using the event facility in the SPE to interrupt the SPU when space is available. The event facility is described in a subsequent section.

Tag-group completion facility

Data movement is performed asynchronously with respect to the execution of the SPU program. Therefore, the CBEA provides a tag-group completion facility so that an application can determine when a specific command or set of commands completes. Independent facilities are available for both the MFC proxy and MFC SPU command queues. A tag ID is provided with each DMA command, and commands issued with the same tag ID to the same queue are called a *tag group*. To determine when tag groups are complete, a mask that selects the tag groups of interest is written to the tag-group mask channel or register. The status of the selected groups is reported in a status register for the MFC proxy command queue. For the MFC SPU command queue, a query type is written to the MFC write tag-group status update request channel. The request can be that either "all selected tag groups are complete" or "any one of the selected groups is complete." Once the query type is set, reading from the blocking tag-group status channel causes the SPU to stall until the query condition is met. Like the command issue sequence, the stall can be avoided by first determining the channel count value of the channel.

The CBEA also provides a way to generate an interrupt when a tag group completes. For the MFC proxy command queue, a similar query type is provided to determine when the interrupt is generated. The interrupt is typically routed to the PPE for processing. For the MFC SPU command queue, the event facility provides the interrupt when the condition is met.

Multisource synchronization facility

In any multiprocessor system, the synchronization of software processes is critical. The PowerPC Architecture provides four instructions (*lwarx*, *ldarx*, *stwcx*, *stdcx*) for performing atomic updates of system memory. The atomic update of system memory is used to create software locks, which are then used to achieve the desired synchronization. Since the SPUs do not have direct access to the main storage domain, the CBEA defines four MFC commands (*getllar*, *putllc*, *putlluc*, *putqlluc*) that allow an SPU to participate in the atomic update of system memory. These commands are known as the *MFC atomic update commands*. They can be issued only by an SPU to the MFC SPU command queue; they cannot be issued by other processors or devices. Unlike the PowerPC instructions, these commands transfer the full reservation granule. In most processors, the reservation granule is typically the same size as the cache line.

The get lock line and reserve (*getllar*) command is similar to the PowerPC *lwarx* and *ldarx* instructions, and the put lock line conditional (*putllc*) command is similar to the *stwcx* and *stdcx* PowerPC instructions. These commands are not queued in the MFC like other MFC commands; they are executed immediately but still require an MFC command queue slot. Since these commands are executed immediately, they do not support the tag parameter, and only one of these commands can be outstanding. The CBEA provides a blocking atomic command status channel for determining when these commands complete and the results of the commands.

The put lock line unconditional (*putlluc*) and put queued lock line unconditional (*putqlluc*) commands are similar to a cacheable PowerPC *store* instruction. These commands are used to unconditionally release a lock. The *putlluc* command is executed immediately, whereas the *putqlluc* command has an implied fence and is placed

in the MFC command queue along with other MFC commands. Since the *putqlluc* command is queued, the tag parameter is supported, and software uses the previously described tag-group completion facility to determine when this command is complete.

A common synchronization method is for software to wait for the value of the lock in system memory to change. In the PowerPC, this requires software to continually poll the memory location associated with the lock. In the CBEA, an SPU event can be generated when a reservation is lost. (The reservation is lost when another processor modifies the memory location associated with the lock.) SPU software can avoid constantly polling the lock by simply obtaining a reservation on the memory location and then waiting for the reservation to be lost using the event facility. Additionally, an SPU interrupt can be generated when the reservation is lost. While the reservation lost event does not guarantee that the lock value has changed, it very effectively eliminates the constant polling of the lock using MFC DMA commands.

Since the performance of atomic updates can suffer when there is a high contention for the lock, this may not be the optimal solution for all synchronization scenarios. Therefore, the CBEA offers additional options for process synchronization, such as the mailbox facility and the signal notification facility. These facilities can be used as alternatives to the atomic update MFC commands, or they can be used in conjunction with them for process synchronization.

Mailbox facility

A key advantage of the CBEA is the independent nature of the SPEs. Since the SPEs are decoupled from the PPE and other SPEs, the CBEA provides a mailbox facility to assist in process-to-process communication and synchronization. The mailbox facility provides a simple, unidirectional communication mechanism typically used by the PPE to send short commands to the SPE and to receive status in return. This facility consists of an inbound, an outbound, and an outbound interrupt mailbox. Each of these mailboxes can have a depth of one or more entries. The direction of the mailbox is relative to the SPU. For example, the inbound mailbox is written by the PPE and read by the SPU. The SPU accesses the mailboxes using the SPU channel instructions. The mailbox channels are blocking, causing the SPU to stall if the outbound mailbox is full or the inbound mailbox is empty. For the PPE and other devices, MMIO registers provide access to the mailboxes and the mailbox status. As the name implies, the outbound interrupt mailbox generates an interrupt when written by the SPU. The interrupt is typically routed to the PPE for processing and is presented as an external exception.

For example, the mailbox facility can be used for a command-driven SPU application. In this example, the SPU is typically stalled waiting for a command to be placed in the inbound mailbox. When a command is received, the SPU performs the requested operation given in the mailbox data or sequence of mailbox data. Once the operation is complete, the SPU places a return code in the outbound interrupt mailbox. The write of the outbound interrupt mailbox generates an interrupt for the PPE that indicates the completion of the requested operation. The SPU code then reads the next command from the inbound mailbox and stalls if a new command is not available. In addition to the application given in this simple example, many other uses were contemplated during the development of this facility.

Signal notification facility

The signal notification facility is very similar to the mailbox facility. It consists of two signal notification registers. From the SPU, these registers are accessed using channels; for other processors, these registers are mapped into the MMIO space. In contrast to the mailbox facility, the signal notification facility is inbound only, and the corresponding channels are only one deep and are blocking. Each of the signal notification registers has two modes of operation, overwrite and logical OR. In the overwrite mode, the contents of the register is replaced with the new value written to the MMIO register even if the SPU has not yet read the current value. In the logical OR mode, the current contents of the signal notification register is ORed with the new value written to the register. In either mode, the signal notification register is set to zero after it is read by the SPU.

Normal MFC "put" commands can be used to access the signal notification facility in another SPE. However, the MFC "send signal" commands are defined for this purpose, allowing an implementation to use a more efficient mechanism than MMIO for signaling between SPUs. The signal notification facility offers an alternative synchronization mechanism between SPUs and, potentially, other devices. It can also be used in conjunction with the other synchronization mechanisms.

For example, the logical OR mode can be used in a task completion notification mechanism. In this example, an SPU assigns a task to another processor; along with each task, the SPU assigns a tag. When the task completes, the processor or device performs the proper synchronization to ensure that the results are visible. It then writes a binary "1" in the signal notification register bit corresponding to the tag. The overwrite mode provides a fast communication when a single bit is not sufficient for synchronization.

SPU event facility and decrementer facility

The SPU event facility provides software with a convenient mechanism to wait for a selected set of conditions. The facility can also interrupt the SPU if one of these conditions occurs. The facility consists of an SPU read event status channel, a channel for reading the SPU write event mask channel, a channel for writing the SPU write event mask channel, and an SPU write event acknowledgment channel. To use the facility, software first writes the SPU write event mask channel to select the conditions of interest. Next, the SPU read event status channel is read. If none of the selected conditions have occurred, the SPU stalls until at least one of the conditions exists. The stall can be avoided either by reading the channel count associated with the SPU write event status channel before reading the channel or by enabling an interrupt to occur when an event status is available. After being notified that an event has occurred, the SPU program should acknowledge the event by writing a binary "1" to the corresponding bit in the SPU write event acknowledgment channel. Acknowledging the event enables the SPU to receive subsequent events for this condition. The SPU read event mask channel is provided for context save and restore operations. Software can also use this channel to eliminate the need for a shadow copy of the currently selected events.

As is apparent in the description of the other facilities, events are defined for many of the stall conditions. The event facility allows software to wait for multiple conditions using a single blocking channel and can cause the SPU to be interrupted. The SPU read event status channel doubles as the interrupt status. The SPU event facility supports the following events:

- MFC tag-group status update event.
- MFC DMA list command stall-and-notify event.
- MFC SPU command queue available event.
- SPU inbound mailbox event.
- SPU outbound mailbox event.
- SPU outbound interrupt mailbox event.
- SPU signal notification 1 event.
- SPU signal notification 2 event.
- Lock line reservation lost event.
- Multisource synchronization event.
- Privileged attention event.
- SPU decrementer event.

Except for the privileged attention event and the SPU decrementer event, these events have been described in the previous sections.

The privileged attention event allows another processor or device to get the attention of an SPU program. Setting the privileged attention bit in the SPU privileged control register causes the privileged attention event in the SPU event facility to be set. The SPU decrementer event is set when the most significant bit of the SPU decrementer changes from zero to one. The SPU software can either wait for these events by reading the SPU read event status channel or enable an interrupt to be generated on the occurrence of any enabled event.

Software management of TLBs facility

Almost all modern processor architectures support a virtual address space that requires the use of an address translation table. The PowerPC, which is no exception, implements virtual addressing using an architected hardware page table in system memory. For performance reasons, processors usually implement a cache of the translations in an on-chip array. In the PowerPC Architecture and the CBEA, this array is called the *TLB*. Processor architectures usually handle a miss of the TLB with software or hardware. In the PowerPC Architecture, the TLB is hardware managed.

In general, caches introduce uncertainty into the performance of a system. For example, a program accessing a cache that does not contain any of the program data (a cold cache) executes more slowly than when accessing a cache that contains the program data (a warm cache). Architectures have addressed this problem for data caches by providing cache management instructions; some even provide software access to the TLBs. Since the PowerPC Architecture does not provide a method to control the contents of the translation cache, there is typically a start-up penalty when a program is first executed that is not acceptable for real-time applications. The CBEA addresses this issue by adding a software TLB management facility. This facility is specified for both the PPE and the SPE. When used in conjunction with hardware management of the TLBs, an operating system can preload or restore an application's translations to reduce the start-up effects.

A second issue with hardware-managed translations is the fixed size and structure of the hardware page table. In addition, all processors within the same operating system partition typically share the same physical hardware page table. With software management, an operating system can choose to manage the TLBs in software for any number of the SPEs or PPEs in a CBEA-compliant processor. This allows an operating system to have a specialized page table structure for an application that is independent of the structure required by the architecture and hardware.

Cache replacement management facility

As mentioned in the previous section, caches introduce uncertainty into the performance of a system. Not only is this uncertainty due to a cold cache at start-up, but it can

also be caused by a characteristic of the application. For example, if a streaming application reads a very long sequential data structure, the contents of the cache is replaced with the infrequently accessed streaming data, thereby creating a cold cache. The cache management instructions offered by architectures cannot prevent the streaming data effects on the cache. Other processor architectures have offered methods to lock data into a cache, but they still do not address the effects of streaming. Since streaming media is a major application area for a CBEA-compliant processor, this issue had to be addressed.

Caches are usually not an architectural data element; thus, the "Cell Broadband Engine Architecture" document [1] focuses only on the need for such a cache replacement management facility. An example in this document guides the processor designers with their implementation choices.

Using the example implementation outlined in [1], an operating system creates a set of replacement management class identifiers (RclassIDs) and an associated replacement management table (RMT). For the PPE, the RclassID is generated from the address of the data accessed using a set of address-range registers. Separate ranges are provided for instruction and data fetches in the PPE. The RclassID is provided as an explicit parameter in the MFC commands.

The RclassID is used as an index into the RMT. The RMT entry selects the sets in the cache (assuming a set-associative cache) that can be replaced if all of the entries in the congruence class are valid and the requested data does not reside in the cache. If the data resides in the cache in a different set, the data simply is accessed. If the data does not exist in the cache and there is a nonvalid entry in the congruence class, an implementation can choose to place the data in the invalid entry even if it does not correspond to the set selected by the RMT.

The cache replacement management facility provides an operating system with two key methods for managing both the data and translation caches, i.e., the TLBs. The first method prevents streaming data from flushing a cache. Accesses to streaming data are tagged with an RclassID that allows the replacement of only a few sets, typically one. This means that the data can be streamed through the cache without replacing older data or translations that may be required by the operating system or application.

The second method allows an operating system to lock data or translations in a cache. This method is beneficial for many types of data and helps to reduce the caching effects inherent in hardware-managed caches. For example, an RTOS can use this facility for a critical PPE interrupt handler.

As mentioned, controlling the interrupt latency is very important for an RTOS. By locking the translations to the key interrupt handlers in the TLBs and locking the interrupt code itself into the caches, an RTOS can reduce the uncertainty of processing an interrupt. Locking data or translations in a cache is achieved by ensuring that only one RMT entry allows for a given cache set to be replaced. In the example above, software should ensure that only the address range of the interrupt routine can select the RMT entry. If these conditions are met, no other access will cause the interrupt hander instructions or translations to be removed from the cache.

To eliminate the start-up cost of the very first interrupt, the RTOS can warm the cache state by pre-touching the interrupt handler when it is first loaded by using the cache management instructions. When used in conjunction with the mediated external exception extension and cache management instructions, the cache replacement management facility essentially eliminates the uncertainty in the execution of an interrupt handler.

The cache replacement management facility is very flexible, providing for both locking data in a cache and reducing the effects of streaming data on caches. While this facility is implementation dependent, the CBEA recommends that an implementation provide an RMT for each major caching structure, including the TLBs.

Internal interrupt controller

Processor architectures typically do not define an interrupt controller. However, in the CBEA all exceptions generated by an SPE are typically handled by the PPEs in the system. The architecture would not be complete if it did not define how these interrupts are presented to the PPE. Therefore, the CBEA defines an internal interrupt controller (IIC) that provides a method to present an interrupt generated by an SPE to the PPE as quickly as possible. The IIC has a sufficient set of features to act as the interrupt controller for the system. If a different feature set is required, the IIC can coexist with a separate system interrupt controller.

The SPU-generated exceptions include the following: internal SPU errors, errors created by the execution of an MFC command, address translation faults, address compare matches, mailbox interrupts, interrupts generated by the execution of an SPU stop-and-signal instruction, and tag-group completion. SPU exceptions cause a PPE external interrupt to occur. The CBEA defines a set of routing registers, interrupt mask registers, and interrupt status registers for each SPE. The routing registers are used to route the interrupt to the PPE that will handle the interrupt. The interrupt mask and status registers are used to enable interrupt conditions and to record the cause of the interrupt. The IIC allows software

to prioritize SPE exceptions with other external interrupts generated by the system.

The interrupt structure in the CBEA is flexible, allowing SPE exceptions to be routed to an IIC or directly to the system interrupt controller. Routing an SPE exception to the system interrupt controller allows for an alternative priority to be employed by the system designer. In addition to handling SPE exceptions, the IIC provides an efficient mechanism for generating interprocessor interrupts. Interprocessor interrupts are generated by a PPE, targeting another PPE in the system.

Resource allocation management

Real-time operating systems were a key consideration during the development of the CBEA and were the primary reason for including features such as cache replacement management and software management of TLBs in the architecture. These features help software maintain real-time guarantees in a single-processor environment, but they do little for the effects created by having multiple processors. In a multiprocessor system, critical resources such as system memory can be consumed by a relatively few devices, which can create latency and bandwidth issues for the remaining devices. To address this issue, the CBEA includes a resource allocation management (RAM) facility.

With RAM enabled, each requester in a CBEA-compliant processor is assigned to a resource allocation group (RAG). A portion of each critical resource is allocated to each RAG. The definition provided in the CBEA ends here. The remaining details of the facility, such as the resources controlled and the controlling mechanism, are highly implementation specific. The rest of this section describes the Cell/B.E. processor implementation of RAM. The Cell/B.E. processor is the first implementation of the CBEA.

In the Cell/B.E. processor, the resources controlled by RAM are system memory and the I/O interfaces. A token-based control mechanism allocates a percentage of the managed resources to each RAG. Before a requester can access a managed resource, a token must be requested from a central token manager. The hypervisor initializes the token manager to generate tokens for each RAG at a rate equivalent to the allocated percentage of the resource. Requesters compete for the tokens generated for their RAG using a round-robin priority scheme. During certain intervals, the frequency of requests from a given RAG may be lower than the allocated token rate; this can result in unused tokens for some RAGs, while other RAGs are throttled by their allocated token rate. The Cell/B.E. design allows the unused tokens to be given to another RAG, which prevents wasting or underutilization of a resource. Typically, software never allocates 100% of a resource in order to account for variations in system performance. This is another source of potential waste. To assist in reducing this waste, the Cell/B.E. processor design allows any RAG to use the unallocated percentage of a resource. The allocation of unused tokens and the unallocated portion of a resource is programmable and can be independently disabled by an operating system.

As mentioned above, 100% of a resource is never allocated in order to account for unexpected performance variations. Allowing unallocated and unused tokens to be used can result in overutilization of a resource. The Cell/B.E. processor implementation addresses this issue by automatically suspending the allocation of these tokens when a resource becomes congested. This is known as *backpressure* from the resource.

At first glance, RAM appears to be a bandwidth management system. However, the bandwidth and latency of system memory are very dependent on the access size and access pattern. For example, the bandwidth is considerably lower if an application continually accesses addresses corresponding to the same physical bank of memory. To avoid this problem, the RAM facility requires a resource to obtain a token for each physical memory bank. Software sets up a single percentage allocation for memory, and the token manager distributes the allocation equally among all memory banks. Therefore, if the memory access pattern of an application is evenly distributed, the achievable memory bandwidth is higher than if the accesses are poorly distributed. An application can also cause poor utilization of system memory if it uses small accesses. Typically, caches prevent small accesses, and they are not a concern for most systems. However, in a CBEA-compliant processor, MFC DMA commands can transfer blocks smaller than a cache line. The Cell/B.E. processor addresses this issue in two ways. First, each token is defined as the transfer of a full cache line, or 128 bytes, regardless of the actual transfer size. Second, the requester is required to obtain two tokens if the transfer size requires a read–modify–write of system memory to update the error-correction code (ECC)—one for the read and one for the write.

Thus, the effective system memory bandwidth is a function of the access pattern and access size. Without the RAM, the accesses performed by one application can affect the system memory bandwidth for all applications. RAM provides an effective mechanism that prevents applications with poor access patterns from affecting the performance of other applications. RAM does not provide a guarantee of memory bandwidth, but it does guarantee access to a resource. Because the effective bandwidth achieved is still dependent on the access pattern, the facility is known as *RAM* instead of *bandwidth reservation*.

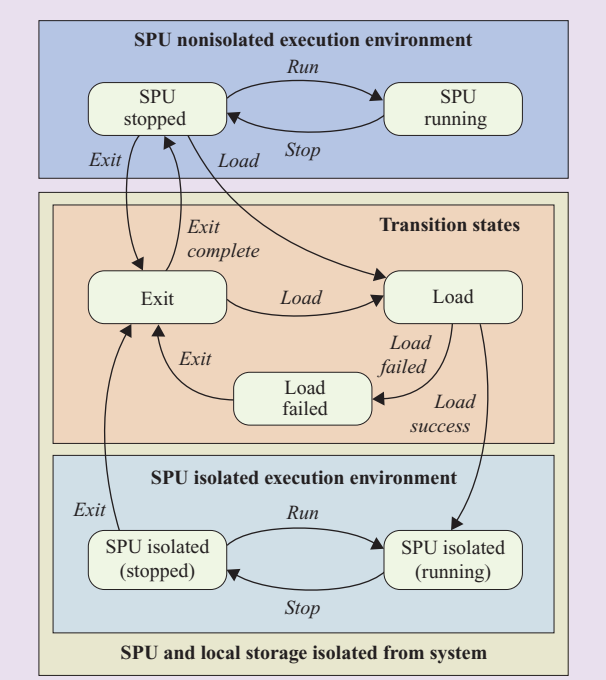

Figure 4

SPU isolated state transitions.

Power management

Power management is always a key consideration in the development of any architecture. Some channels in the SPE are defined as blocking, which essentially allows an SPU to enter a very low-power state while waiting for the channel to become available. The CBEA also defines the following set of more traditional power management states: active, slow(n), pause(n), state retained and isolated (SRI), and state lost and isolated (SLI).

Each element in a CBEA-compliant processor can be in any one of the five defined states. In the active state, the element runs at full performance level. No power management is applied except for the dynamic power management provided by blocking channels or the gating of nonactive circuits found in most advanced designs.

Several levels are allowed for the slow(n) state, with a higher value of n representing more aggressive power savings. As the name implies, the element executes at a slower rate; thus, the performance is degraded but still guaranteed to make forward progress. An example of a slow state is running an element at a slower clock frequency to save power. The pause(n) state is similar to the slow(n) state except that forward progress is not guaranteed. A simple example of a pause state is the situation in which an SPU is stopped but the MFC is still active.

In the three power management states described above, the context of the element is maintained. For example, the TLBs and caches remain coherent with system memory and other elements in the system. In the SRI and SLI states, the element no longer interacts with other elements in the system; thus, the full context for the element cannot be maintained. (These power management states should not be confused with the SPU isolation facility described in the next section.) Software must prepare the element before entering or leaving these states. The SRI state allows software to keep the bulk of the context while saving most of the power of the element. An example of the SRI state is the case in which the SPE is isolated from the system and the contents of the local storage and MMIO registers are maintained. Coherency of the TLBs and caches is not maintained. The SLI state is the most aggressive power-saving state. In this state, nothing is maintained, including the local storage, registers, and caches. An example of this state is the situation in which power is removed from an SPE or PPE. The actual implementation and supported power management states are implementation specific. Therefore, the CBEA does not define the details.

Isolated execution environment

The CBEA includes an SPU isolation facility that is used to create a secure execution environment within an SPE. **Figure 4** is a state diagram illustrating the transitions into and out of the SPU isolated execution environment. To enter an SPU isolated execution environment, a pointer to the code to execute in the secure execution environment is loaded into the SPU signal notification registers 1 and 2, and the SPU is started using a *Load* request. When it receives the *Load* request, the SPE first sets up the isolated execution environment by removing access to a portion of the local storage. The remaining portion of local storage remains open. All accesses to the local storage alias and MFC DMA commands are forced into the open area. After the isolated environment is initialized, the SPE loads the application into the isolated area of local storage. As it is loaded, the code is authenticated and decrypted by using a hardware root key. If the load succeeds, the application is executed. Once in the SPU isolated execution environment, the only way out is to stop the SPU and restart it with an *Exit* request. If the *Load* fails, the SPU remains halted in a transition state. The only way out of the transition state is to restart the SPU by using either another *Load* request or an *Exit* request.

In addition to the isolated execution environment, the SPU isolation facility provides an application with a random number generator and persistent storage. The

persistent storage maintains its state between isolation sessions and can be accessed only by an application executing in the SPU isolated execution environment.

Software development toolkit

To provide software developers exposure to the CBEA with and without access to hardware, a Software Development Kit (SDK) is available on the IBM alphaWorks* Web site (*www.alphaworks.ibm.com/topics/cell*) with open-source content distributed on the Barcelona Supercomputing Center Web site (*www.bsc.es/projects/deepcomputing/linuxoncell*).

The SDK is continuously being enhanced with additional components and features. The key foundational components of the SDK are as follows:

- A Cell/B.E. processor–enabled Linux kernel supporting the unique CBEA features.
- The IBM full system simulator for the Cell/B.E. processor that supports simulation of either uniprocessor or dual-processor systems. Both functional and cycle accurate simulation modes are provided so that programmers can maximize either interactivity or timing accuracy.
- A system root image (sysroot) that provides a standard set of Linux operating system utilities and services for use in the simulated Linux operating system environment.
- Standard CBEA-specific libraries including the newlib C languages standard library, SIMD math libraries, and an SPE runtime management library that exposes the SPEs as heterogeneous POSIX threads.
- Code-generation tools including GNU and IBM XL C/C++ compilers. The compilers support the automatic generation of SIMD code for both the PPE and the SPE.
- Productivity tools such as an Eclipse-based integration development environment (IDE). The IDE provides an integration framework for building, deploying, and managing CBEA software.
- Cell/B.E. processor–enabled debug tools including the GNU debugger (gdb). The debugger has been enhanced to enable combined PPU and SPU debugging and to provide user access to the unique states of the Cell/B.E. processor.
- Performance analysis tools including the OProfile system profiling tool, the Cell/B.E. processor performance counter, which provides access to Cell/B.E. processor performance monitoring facilities, code analyzer, Feedback-Directed Program Restructuring technology of FDPR-pro, and a static performance analysis tool called *spu_timing*. The output from many of the tools can be visualized using the Eclipse-based visual performance analyzer (VPA).
- An SPE executive for invoking SPU executables directly from a Linux shell. The executive provides standard system services including file I/O, shared memory, memory map, and time of day.
- Many code samples that demonstrate programming techniques, optimized libraries with source code to jump-start application development, and workloads to demonstrate the computational capabilities of the Cell/B.E. processor.
- Programming model frameworks such as the Accelerated Library Framework (ALF), which provides services to assist in the development of parallel applications and libraries on multicore architectures with hierarchical memory such as the CBEA.

The SDK is supported on the x86, PowerPC 64, and Cell/B.E. processor blade systems running Fedora** core for the Linux operating system. These configurations minimize system costs, ensuring wide access to anyone who wants to evaluate Cell/B.E. technology, learn Cell/B.E. programming, or develop Cell/B.E. applications and tools.

Software standards for CBEA

Critical to the success of the CBEA is the adoption of software standards and conformance to these standards. From the outset, the Sony Group, Toshiba, and IBM worked together to establish an initial set of standards including the following:

- Application binary interface (ABI) specifications including the *SPU Application Binary Interface Specification* and the *CBE Linux Reference Implementation ABI* specification.
- Language specifications including the *SPU Assembly Language Specification* and the *C/C++ Language Extensions for Cell Broadband Engine Architecture* specification.
- Standardized library specifications including the SPE Runtime Management Library and the SIMD Vector Math Library specifications.

Adherence to these specifications ensures that application code is portable and that tools produce binaries that will coexist. To ensure compliance with the standards, conformance tests have been developed and made generally available.

Conclusions: The power of the Cell/B.E. processor

The real power behind the CBEA is the optimization made possible by the heterogeneous design and the memory movement, provided by the DMA, for hiding memory latency. While initially directed toward media and streaming applications, the flexibility of the architecture allows for a multitude of application and programming techniques. For example, several papers describe CBEA applications that include features such as software-managed caches implemented in the local storage and software threading of the SPEs [8]. Additionally, a generalized implementation of a software cache is provided in the SDK.

A formal request for change (RFC) process governs changes and additions. All RFCs require unanimous acceptance by the three partner companies—the Sony Group, Toshiba, and IBM. Each of the partner companies serves as an advocate to ensure that its business partners are represented in the RFC process.

The CBEA offers a wide variety of features to improve the performance of systems requiring RTOS, streaming, and computing applications. In addition, the CBEA defines a unique approach to security by offering an SPU isolation facility. When used properly, this facility provides an isolated execution environment that is not accessible from other elements or by external means. The features and flexibility of the CBEA combined with the new development environment and software standards create a processor architecture and ecosystem poised to set the standard for application acceleration.

Acknowledgments

The CBEA and the Cell/B.E. processor (the first implementation of the architecture) resulted from a deep collaboration of engineers from Sony Computer Entertainment Incorporated, Toshiba, and IBM. Each of the partners brought to the team a unique set of requirements, experience, and perspectives. The diversity of the team enabled it to look beyond the traditional processor architectures and develop the heterogeneous processing structure provided by the CBEA. More than five years and countless person-hours went into the development of the CBEA and the Cell/B.E. processor. The authors also wish to thank the anonymous reviewers for their suggestions for improving the content of this paper.

References

1. Cell Broadband Engine Architecture; see *http://www.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA2776387257060006E61BA*.
2. J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell Multiprocessor," *IBM J. Res. & Dev.* **49**, No. 4/5, 589–604 (2005).
3. PowerPC Architecture Book, Version 2.02; see *http://www.ibm.com/developerworks/power/library/pa-archguidev2/*.
4. Synergistic Processor Unit Instruction Set Architecture; see *http://www.ibm.com/chips/techlib/techlib.nsf/techdocs/76CA6C7304210F3987257060006F2C44*.
5. B. Flachs, S. Asano, S. H. Dhong, H. P. Hofstee, G. Gervais, R. Kim, T. Le, et al., "The Microarchitecture of the Synergistic Processor for a Cell Processor," *IEEE J. Solid-State Circuits* **41**, No. 1, 63–70 (2006).
6. D. A. Brokenshire, "Maximizing the Power of the Cell Broadband Engine Processor: 25 Tips to Optimal Application Performance"; see *http://www.ibm.com/developerworks/power/library/pa-celltips1/*.
7. D. Krolak, "Just Like Being There: Papers from the Fall Processor Forum 2005: Unleashing the Cell Broadband Engine Processor: The Element Interconnect Bus"; see *http://www.ibm.com/developerworks/power/library/pa-fpfeib/index.html*.
8. C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich, "Ray Tracing on the Cell Processor"; see *http://graphics.cs.uni-sb.de/~benthin/cellrt06.pdf*.

Received July 22, 2006; accepted for publication March 1, 2007; Internet publication August 9, 2007

Charles R. Johns *IBM Systems and Technology Group, 11400 Burnet Road, Austin, Texas 78758 (crjohns@us.ibm.com).*
Mr. Johns is a Senior Technical Staff Member in the Sony/Toshiba/IBM Design Center. He received his B.S. degree in electrical engineering from the University of Texas at Austin in 1984. After joining IBM Austin in 1984, Mr. Johns worked on various disk, memory, voice communication, and graphics adapters for the IBM Personal Computer. From 1988 until he transferred to the STI project in 2000, he was part of the graphics organization and was responsible for the architecture and development of entry and midrange 3D graphics adapters and raster engines. Mr. Johns is now responsible for the CBEA; he participated in the development of the Cell/B.E. processor, which is the first implementation of the CBEA. Mr. Johns is an IBM Master Inventor.

Daniel A. Brokenshire *IBM Systems and Technology Group, 11400 Burnet Road, Austin, Texas 78758 (brokensh@us.ibm.com).*
Mr. Brokenshire is a Senior Technical Staff Member with six years of experience in the Cell/B.E. Processor Design Center. He currently serves as a senior member of the IBM Cell/B.E. Processor Systems Enablement team working on the Cell/B.E. processor SDK. His responsibilities include the development of programming standards, language extensions, reusable software libraries, and software documentation. Mr. Brokenshire received a B.S. degree in computer science and B.S. and M.S. degrees in electrical engineering, all from Oregon State University. Prior to his work on the Cell/B.E. processor, he enjoyed a productive career developing 3D graphics products for Tektronix, Inc., and IBM.