Cell Multiprocessor Communication Network: Built for Speed

Multicore designs promise various power-performance and area-performance benefits. But inadequate design of the on-chip communication network can deprive applications of these benefits. To illuminate this important point in multicore processor design, the authors analyze the Cell processor's communication network, using a series of benchmarks involving various DMA traffic patterns and synchronization protocols.

Michael Kistler
IBM Austin Research Laboratory

Michael Perrone
IBM TJ Watson Research Center

Fabrizio Petrini
Pacific Northwest National Laboratory

Over the past decade, high-performance computing has ridden the wave of commodity computing, building cluster-based parallel computers that leverage the tremendous growth in processor performance fueled by the commercial world. As this pace slows, processor designers face complex problems in their efforts to increase gate density, reduce power consumption, and design efficient memory hierarchies. Processor developers are looking for solutions that can keep up with the scientific and industrial communities' insatiable demand for computing capability and that also have a sustainable market outside science and industry.

A major trend in computer architecture is integrating system components onto the processor chip. This trend is driving the development of processors that can perform functions typically associated with entire systems. Building modular processors with multiple cores is far more cost-effective than building monolithic processors, which are prohibitively expensive to develop, have high power consumption, and give limited return on investment. Multicore system-on-chip (SoC) processors integrate several identical, independent processing units on the same die, together with network interfaces, acceleration units, and other specialized units.

Researchers have explored several design avenues in both academia and industry. Examples include MIT's Raw multiprocessor, the University of Texas's Trips multiprocessor, AMD's Opteron, IBM's Power5, Sun's Niagara, and Intel's Montecito, among many others. (For details on many of these processors, see the March/April 2005 issue of *IEEE Micro*.)

In all multicore processors, a major technological challenge is designing the internal, on-chip communication network. To realize the unprecedented computational power of the many available processing units, the network must provide very high performance in

Published by the IEEE Computer Society

latency and in bandwidth. It must also resolve contention under heavy loads, provide fairness, and hide the processing units' physical distribution as completely as possible.

Another important dimension is the nature and semantics of the communication primitives available for interactions between the various processing units. Pinkston and Shin have recently compiled a comprehensive survey of multicore processor design challenges, with particular emphasis on internal communication mechanisms.[1]

The Cell Broadband Engine processor (known simply as the Cell processor), jointly developed by IBM, Sony, and Toshiba, uses an elegant and natural approach to on-chip communication. Relying on four slotted rings coordinated by a central arbiter, it borrows a mainstream communication model from high-performance networks in which processing units cooperate through remote direct memory accesses (DMAs).[2] From functional and performance viewpoints, the on-chip network is strikingly similar to high-performance networks commonly used for remote communication in commodity computing clusters and custom supercomputers.

In this article, we explore the design of the Cell processor's on-chip network and provide insight into its communication and synchronization protocols. We describe the various steps of these protocols, the algorithms involved, and their basic costs. Our performance evaluation uses a collection of benchmarks of increasing complexity, ranging from basic communication patterns to more demanding collective patterns that expose network behavior under congestion.

Design rationale

The Cell processor's design addresses at least three issues that limit processor performance: memory latency, bandwidth, and power.

Historically, processor performance improvements came mainly from higher processor clock frequencies, deeper pipelines, and wider issue designs. However, memory access speed has not kept pace with these improvements, leading to increased effective memory latencies and complex logic to hide them. Also, because complex cores don't allow a large number of concurrent memory accesses, they underutilize execution pipelines and memory bandwidth, resulting in poor chip area use and increased power dissipation without commensurate performance gains.[3]

For example, larger memory latencies increase the amount of speculative execution required to maintain high processor utilization. Thus, they reduce the likelihood that useful work is being accomplished and increase administrative overhead and bandwidth requirements. All of these problems lead to reduced power efficiency.

Power use in CMOS processors is approaching the limits of air cooling and might soon begin to require sophisticated cooling techniques.[4] These cooling requirements can significantly increase overall system cost and complexity. Decreasing transistor size and correspondingly increasing subthreshold leakage currents further increase power consumption.[5]

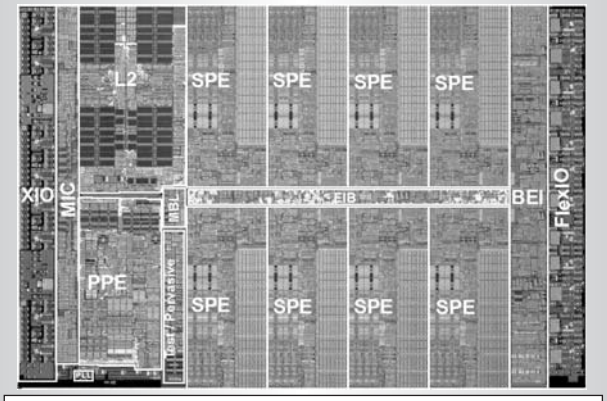
Performance improvements from further increasing processor frequencies and pipeline depths are also reaching their limits.[6] Deeper pipelines increase the number of stalls from data dependencies and increase branch misprediction penalties.

The Cell processor addresses these issues by attempting to minimize pipeline depth, increase memory bandwidth, allow more simultaneous, in-flight memory transactions, and improve power efficiency and performance.[7] These design goals led to the use of flexible yet simple cores that use area and power efficiently.

Processor overview

The Cell processor is the first implementation of the Cell Broadband Engine Architetcture (CBEA), which is a fully compatible extension of the 64-bit PowerPC Architecture. Its initial target is the PlayStation 3 game console, but its capabilities also make it well suited for other applications such as visualization, image and signal processing, and various scientific and technical workloads.

Figure 1 shows the Cell processor's main functional units. The processor is a heterogeneous, multicore chip capable of massive floating-point processing optimized for computation-intensive workloads and rich broadband media applications. It consists of one 64-bit power processor element (PPE), eight specialized coprocessors called synergistic processor elements (SPEs), a high-speed mem-



BEI	Broadband engine interface	MBL	MIC bus logic
EIB	Element interconnect bus	PPE	Power processor element
FlexIO	High-speed I/O interface	SPE	Synergistic processor element
L2	Level 2 cache	XIO	Extreme data rate I/O cell
MIC	Memory interface controller		Test control unit/pervasive logic

Figure 1. Main functional units of the Cell processor.

ory controller, and a high-bandwidth bus interface, all integrated on-chip. The PPE and SPEs communicate through an internal high-speed element interconnect bus (EIB).

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 EIB supports 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 memory interface controller (MIC) provides a peak bandwidth of 25.6 Gbytes/s to main memory. The I/O controller provides peak bandwidths of 25 Gbytes/s inbound and 35 Gbytes/s outbound.

The PPE, the Cell's main processor, runs the operating system and coordinates the SPEs. It is a traditional 64-bit PowerPC processor core with a vector multimedia extension (VMX) unit, 32-Kbyte level 1 instruction and data caches, and a 512-Kbyte level 2 cache. The PPE is a dual-issue, in-order-execution design, with two-way simultaneous multithreading.

Each SPE consists of a synergistic processor unit (SPU) and a memory flow controller (MFC). The MFC includes a DMA controller, a memory management unit (MMU), a bus interface unit, and an atomic unit for synchronization with other SPUs and the PPE. The SPU is a RISC-style processor with an instruction set and a microarchitecture designed for high-performance data streaming and data-intensive computation. The SPU includes a 256-Kbyte local-store memory to hold an SPU program's instructions and data. The SPU cannot access main memory directly, but it can issue DMA commands to the MFC to bring data into local store or write computation results back to main memory. The SPU can continue program execution while the MFC independently performs these DMA transactions. No hardware data-load prediction structures exist for local store management, and each local store must be managed by software.

The MFC performs DMA operations to transfer data between local store and system memory. DMA operations specify system memory locations using fully compliant PowerPC virtual addresses. DMA operations can transfer data between local store and any resources connected via the on-chip interconnect (main memory, another SPE's local store, or an I/O device). Parallel SPE-to-SPE transfers are sustainable at a rate of 16 bytes per SPE clock, whereas aggregate main-memory bandwidth is 25.6 Gbytes/s for the entire Cell processor.

Each SPU has 128 128-bit single-instruction, multiple-data (SIMD) registers. The large number of architected registers facilitates highly efficient instruction scheduling and enables important optimization techniques such as loop unrolling. All SPU instructions are inherently SIMD operations that the pipeline can run at four granularities: 16-way 8-bit integers, eight-way 16-bit integers, four-way 32-bit integers or single-precision floating-point numbers, or two 64-bit double-precision floating-point numbers.

The SPU is an in-order processor with two instruction pipelines, referred to as the *even* and *odd* pipelines. The floating- and fixed-point units are on the even pipeline, and the rest of the functional units are on the odd pipeline. Each SPU can issue and complete up to two instructions per cycle—one per pipeline. The SPU can approach this theoretical limit for a wide variety of applications. All single-precision operations (8-bit, 16-bit, or 32-bit integers or 32-bit floats) are fully pipelined and can be issued at the full SPU clock rate (for example, four 32-bit floating-

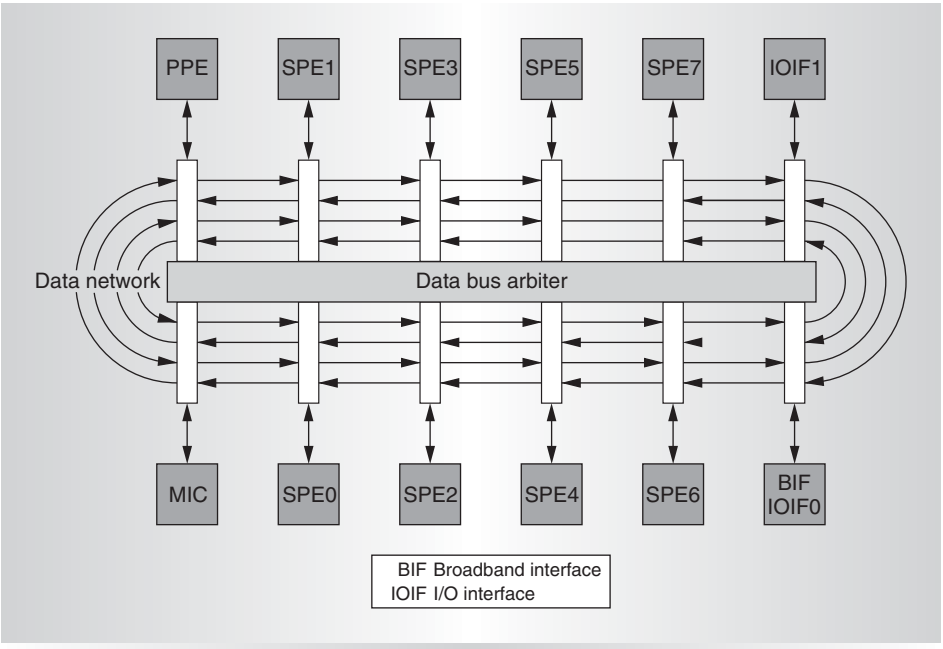


Figure 2. Element interconnect bus (EIB).

point operations per SPU clock cycle). The two-way double-precision floating-point operation is partially pipelined, so its instructions issue at a lower rate (two double-precision flops every seven SPU clock cycles). When using single-precision floating-point fused multiply-add instructions (which count as two operations), the eight SPUs can perform a total of 64 operations per cycle.

Communication architecture

To take advantage of all the computation power available on the Cell processor, work must be distributed and coordinated across the PPE and the SPEs. The processor's specialized communication mechanisms allow efficient data collection and distribution as well as coordination of concurrent activities across the computation elements. Because the SPU can act directly only on programs and data in its own local store, each SPE has a DMA controller that performs high-bandwidth data transfer between local store and main memory. These DMA engines also allow direct transfers between the local stores of two SPUs for pipeline or producer-consumer-style parallel applications.

At the other end of the spectrum, the SPU can use either signals or mailboxes to perform simple low-latency signaling to the PPE or other SPEs. Supporting more-complex synchronization mechanisms is a set of atomic operations available to the SPU, which operate in a similar manner as the PowerPC architecture's lwarx/stwcx atomic instructions. In fact, the SPU's atomic operations interoperate with PPE atomic instructions to build locks and other synchronization mechanisms that work across the SPEs and the PPE. Finally, the Cell allows memory-mapped access to nearly all SPE resources, including the entire local store. This provides a convenient and consistent mechanism for special communications needs not met by the other techniques.

The rich set of communications mechanisms in the Cell architecture enables programmers to efficiently implement widely used programming models for parallel and distributed applications. These models include the function-offload, device-extension, computational-acceleration, streaming, shared-memory-multiprocessor, and asymmetric-thread-runtime models.[8]

Element interconnect bus

Figure 2 shows the EIB, the heart of the Cell processor's communication architecture, which enables communication among the PPE, the SPEs, main system memory, and external I/O. The EIB has separate commu-

nication paths for commands (requests to transfer data to or from another element on the bus) and data. Each bus element is connected through a point-to-point link to the address concentrator, which receives and orders commands from bus elements, broadcasts the commands in order to all bus elements (for snooping), and then aggregates and broadcasts the command response. The command response is the signal to the appropriate bus elements to start the data transfer.

The EIB data network consists of four 16-byte-wide data rings: two running clockwise, and the other two counterclockwise. Each ring potentially allows up to three concurrent data transfers, as long as their paths don't overlap. To initiate a data transfer, bus elements must request data bus access. The EIB data bus arbiter processes these requests and decides which ring should handle each request. The arbiter always selects one of the two rings that travel in the direction of the shortest transfer, thus ensuring that the data won't need to travel more than halfway around the ring to its destination. The arbiter also schedules the transfer to ensure that it won't interfere with other in-flight transactions. To minimize stalling on reads, the arbiter gives priority to requests coming from the memory controller. It treats all others equally in round-robin fashion. Thus, certain communication patterns will be more efficient than others.

The EIB operates at half the processor-clock speed. Each EIB unit can simultaneously send and receive 16 bytes of data every bus cycle. The EIB's maximum data bandwidth is limited by the rate at which addresses are snooped across all units in the system, which is one address per bus cycle. Each snooped address request can potentially transfer up to 128 bytes, so in a 3.2GHz Cell processor, the theoretical peak data bandwidth on the EIB is 128 bytes × 1.6 GHz = 204.8 Gbytes/s.

The on-chip I/O interfaces allow two Cell processors to be connected using a coherent protocol called the broadband interface (BIF), which effectively extends the multiprocessor network to connect both PPEs and all 16 SPEs in a single coherent network. The BIF protocol operates over IOIF0, one of the two available on-chip I/O interfaces; the other interface, IOIF1, operates only in noncoherent mode. The IOIF0 bandwidth is configurable, with a peak of 30 Gbytes/s outbound and 25 Gbytes/s inbound.

The actual data bandwidth achieved on the EIB depends on several factors: the destination and source's relative locations, the chance of a new transfer's interfering with transfers in progress, the number of Cell chips in the system, whether data transfers are to/from memory or between local stores in the SPEs, and the data arbiter's efficiency.

Reduced bus bandwidths can result in the following cases:

- All requestors access the same destination, such as the same local store, at the same time.
- All transfers are in the same direction and cause idling on two of the four data rings.
- A large number of partial cache line transfers lowers bus efficiency.
- All transfers must travel halfway around the ring to reach their destinations, inhibiting units on the way from using the same ring.

Memory flow controller

Each SPE contains an MFC that connects the SPE to the EIB and manages the various communication paths between the SPE and the other Cell elements. The MFC runs at the EIB's frequency—that is, at half the processor's speed. The SPU interacts with the MFC through the SPU channel interface. Channels are unidirectional communication paths that act much like first-in first-out fixed-capacity queues. This means that each channel is defined as either read-only or write-only from the SPU's perspective. In addition, some channels are defined with blocking semantics, meaning that a read of an empty read-only channel or a write to a full write-only channel causes the SPU to block until the operation completes. Each channel has an associated count that indicates the number of available elements in the channel. The SPU uses the read channel (rdch), write channel (wrch), and read channel count (rchcnt) assembly instructions to access the SPU channels.

DMA. The MFC accepts and processes DMA commands that the SPU or the PPE issued using the SPU channel interface or memory-mapped I/O (MMIO) registers. DMA commands queue in the MFC, and the SPU or PPE

(whichever issued the command) can continue execution in parallel with the data transfer, using either polling or blocking interfaces to determine when the transfer is complete. This autonomous execution of MFC DMA commands allows convenient scheduling of DMA transfers to hide memory latency.

The MFC supports naturally aligned transfers of 1, 2, 4, or 8 bytes, or a multiple of 16 bytes to a maximum of 16 Kbytes. DMA list commands can request a list of up to 2,048 DMA transfers using a single MFC DMA command. However, only the MFC's associated SPU can issue DMA list commands. A DMA list is an array of DMA source/destination addresses and lengths in the SPU's local storage. When an SPU issues a DMA list command, the SPU specifies the address and length of the DMA list in the local store.[9] Peak performance is achievable for transfers when both the effective address and the local storage address are 128-byte aligned and the transfer size is an even multiple of 128 bytes.

Signal notification and mailboxes. The signal notification facility supports two signaling channels: Sig_Notify_1 and Sig_Notify_2. The SPU can read its own signal channels using the read-blocking SPU channels SPU_RdSigNotify1 and SPU_RdSigNotify2. The PPE or an SPU can write to these channels using memory-mapped addresses. A special feature of the signaling channels is that they can be configured to treat writes as logical OR operations, allowing simple but powerful collective communication across processors.

Each SPU also has a set of mailboxes that can function as a narrow (32-bit) communication channel to the PPE or another SPE. The SPU has a four-entry, read-blocking inbound mailbox and two single-entry, write-blocking outbound mailboxes, one of which will also generate an interrupt to the PPE when the SPE writes to it. The PPE uses memory-mapped addresses to write to the SPU's inbound mailbox and read from either of the SPU's outbound mailboxes. In contrast to the signal notification channels, mailboxes are much better suited for one-to-one communication patterns such as master-slave or producer-consumer models. A typical round-trip communication using mailboxes between two SPUs takes approximately 300 nanoseconds (ns).

Atomic operations. To support more complex synchronization mechanisms, the SPU can use special DMA operations to atomically update a lock line in main memory. These operations, called get-lock-line-and-reserve (getllar) and put-lock-line-conditional (putllc), are conceptually equivalent to the PowerPC load-and-reserve (lwarx) and store-conditional (stcwx) instructions.

The getllar operation reads the value of a synchronization variable in main memory and sets a reservation on this location. If the PPE or another SPE subsequently modifies the synchronization variable, the SPE loses its reservation. The putllc operation updates the synchronization variable only if the SPE still holds a reservation on its location. If putllc fails, the SPE must reissue getllar to obtain the synchronization variable's new value and then retry the attempt to update it with another putllc. The MFC's atomic unit performs the atomic DMA operations and manages reservations held by the SPE.

Using atomic updates, the SPU can participate with the PPE and other SPUs in locking protocols, barriers, or other synchronization mechanisms. The atomic operations available to the SPU also have some special features, such as notification through an interrupt when a reservation is lost, that enable more efficient and powerful synchronization than traditional approaches.

Memory-mapped I/O (MMIO) resources. Memory-mapped resources play a role in many of the communication mechanisms already discussed, but these are really just special cases of the Cell architecture's general practice of making all SPE resources available through MMIO. These resources fall into four broad classes:

- *Local storage*. All of an SPU's local storage can be mapped into the effective-address space. This allows the PPE to access the SPU's local storage with simple loads and stores, though doing so is far less efficient than using DMA. MMIO access to local storage is not synchronized with SPU execution, so programmers must ensure that the SPU program is designed to allow unsynchronized access to its data (for example, by using the "volatile" variables) when exploiting this feature.

- *Problem state memory map*. Resources in this class, intended for use directly by application programs, include access to the SPE's DMA engine, mailbox channels, and signal notification channels.
- *Privilege 1 memory map*. These resources are available to privileged programs such as the operating system or authorized subsystems to monitor and control the execution of SPU applications.
- *Privilege 2 memory map*. The operating system uses these resources to control the resources available to the SPE.

DMA flow

The SPE's DMA engine handles most communications between the SPU and other Cell elements and executes DMA commands issued by either the SPU or the PPE. A DMA command's data transfer direction is always referenced from the SPE's perspective. Therefore, commands that transfer data into an SPE (from main storage to local store) are considered get commands (gets), and transfers of data out of an SPE (from local store to main storage) are considered put commands (puts).

DMA transfers are coherent with respect to main storage. Programmers should be aware that the MFC might process the commands in the queue in a different order from that in which they entered the queue. When order is important, programmers must use special forms of the get and put commands to enforce either barrier or fence semantics against other commands in the queue.

The MFC's MMU handles address translation and protection checking of DMA accesses to main storage, using information from page and segment tables defined in the PowerPC architecture. The MMU has a built-in translation look-aside buffer (TLB) for caching the results of recently performed translations.

The MFC's DMA controller (DMAC) processes DMA commands queued in the MFC. The MFC contains two separate DMA command queues:

- *MFC SPU command queue*, for commands issued by the associated SPU using the channel interface; and
- *MFC proxy command queue*, for commands issued by the PPE or other devices using MMIO registers.

The Cell architecture doesn't dictate the size of these queues and recommends that software not assume a particular size. This is important to ensure functional correctness across Cell architecture implementations. However, programs should use DMA queue entries efficiently because attempts to issue DMA commands when the queue is full will lead to performance degradation. In the Cell processor, the MFC SPU command queue contains 16 entries, and the MFC proxy command queue contains eight entries.

Figure 3 illustrates the basic flow of a DMA transfer to main storage initiated by an SPU. The process consists of the following steps:

1. The SPU uses the channel interface to place the DMA command in the MFC SPU command queue.
2. The DMAC selects a command for processing. The set of rules for selecting the command for processing is complex, but, in general, a) commands in the SPU command queue take priority over commands in the proxy command queue, b) the DMAC alternates between get and put commands, and c) the command must be ready (not waiting for address resolution or list element fetch or dependent on another command).
3. If the command is a DMA list command and requires a list element fetch, the DMAC queues a request for the list element to the local-store interface. When the list element is returned, the DMAC updates the DMA entry and must reselect it to continue processing.
4. If the command requires address translation, the DMAC queues it to the MMU for processing. When the translation is available in the TLB, processing proceeds to the next step (unrolling). On a TLB miss, the MMU performs the translation, using the page tables stored in main memory, and updates the TLB. The DMA entry is updated and must be reselected for processing to continue.
5. Next, the DMAC unrolls the command—that is, creates a bus request to transfer the next block of data for the command. This bus request can transfer up to 128 bytes of data but can transfer less, depending on alignment issues or the amount of data the

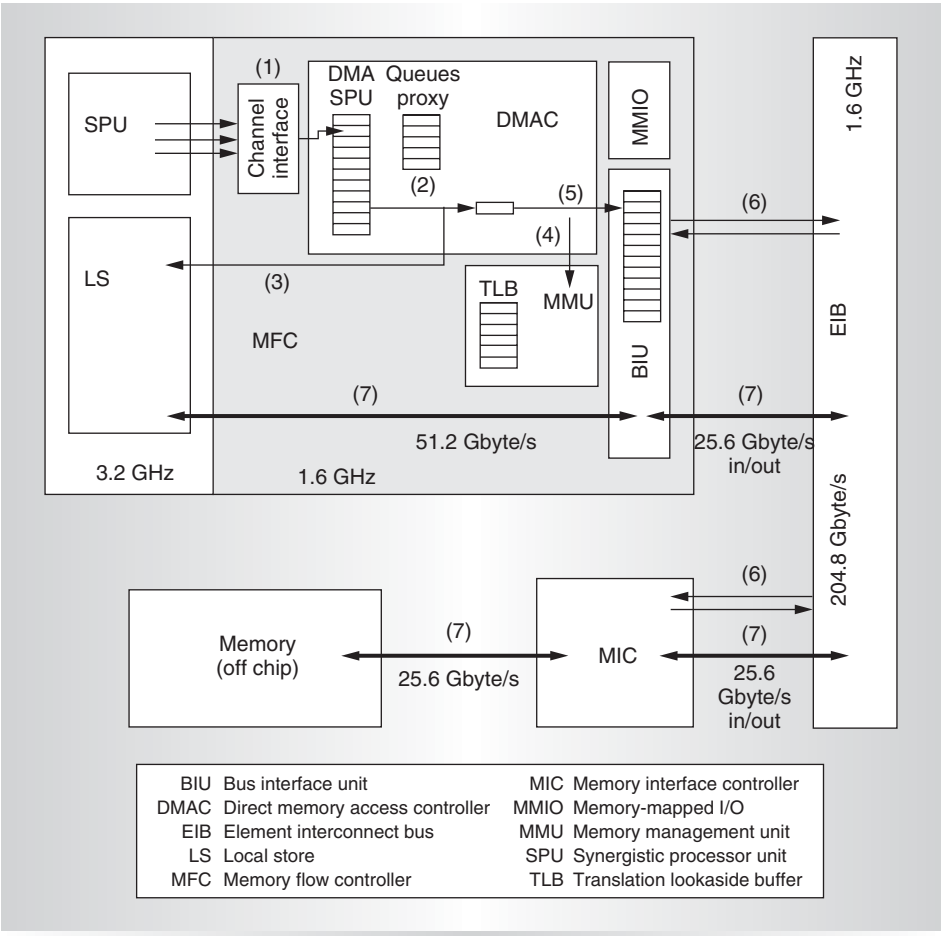


Figure 3. Basic flow of a DMA transfer.

DMA command requests. The DMAC then queues this bus request to the bus interface unit (BIU).

6. The BIU selects the request from its queue and issues the command to the EIB. The EIB orders the command with other outstanding requests and then broadcasts the command to all bus elements. For transfers involving main memory, the MIC acknowledges the command to the EIB which then informs the BIU that the command was accepted and data transfer can begin.
7. The BIU in the MFC performs the reads to local store required for the data transfer. The EIB transfers the data for this request between the BIU and the MIC. The MIC transfers the data to or from the off-chip memory.
8. The unrolling process produces a sequence of bus requests for the DMA command, that pipeline through the communication network. The DMA command remains in the MFC SPU command queue until all its bus requests have completed. However, the DMAC can continue to process other DMA commands. When all bus requests for a command have completed, the DMAC signals command completion to the SPU and removes the command from the queue.

In the absence of congestion, a thread running on the SPU can issue a DMA request in as little as 10 clock cycles—the time needed to write to the five SPU channels that describe the source and destination addresses, the DMA size, the DMA tag, and the DMA command. At that point, the DMAC can process the DMA request without SPU intervention.

The overall latency of generating the DMA command, initially selecting the command,

and unrolling the first bus request to the BIU—or, in simpler terms, the flow-through latency from SPU issue to injection of the bus request into the EIB—is roughly 30 SPU cycles when all resources are available. If list element fetch is required, it can add roughly 20 SPU cycles. MMU translation exceptions by the SPE are very expensive and should be avoided if possible. If the queue in the BIU becomes full, the DMAC is blocked from issuing further requests until resources become available again.

A transfer's command phase involves snooping operations for all bus elements to ensure coherence and typically requires some 50 bus cycles (100 SPU cycles) to complete. For gets, the remaining latency is attributable to the data transfer from off-chip memory to the memory controller and then across the bus to the SPE, which writes it to local store. For puts, DMA latency doesn't include transferring data all the way to off-chip memory because the SPE considers the put complete once all data have been transferred to the memory controller.

Experimental results

We conducted a series of experiments to explore the major performance aspects of the Cell's on-chip communication network, its protocols, and the pipelined communication's impact. We developed a suite of microbenchmarks to analyze the internal interconnection network's architectural features. Following the research path of previous work on traditional high-performance networks,[2] we adopted an incremental approach to gain insight into several aspects of the network. We started our analysis with simple pairwise, congestion-free DMAs, and then we increased the benchmarks' complexity by including several patterns that expose the contention resolution properties of the network under heavy load.

Methodology

Because of the limited availability of Cell boards at the time of our experiments, we performed most of the software development on IBM's Full-System Simulator for the Cell Broadband Engine Processor (available at http://www-128.ibm.com/developerworks/power/cell/).[10,11] We collected the results presented here using an experimental evaluation board at the IBM Research Center in Austin. The Cell processor on this board was running at 3.2 GHz. We also obtained results from an internal version of the simulator that includes performance models for the MFC, EIB, and memory subsystems. Performance simulation for the Cell processor is still under development, but we found good correlation between the simulator and hardware in our experiments. The simulator let us observe aspects of system behavior that would be difficult or practically impossible to observe on actual hardware.

We developed the benchmarks in C, using several Cell-specific libraries to orchestrate activities between the PPE and various SPEs. In all tests, the DMA operations are issued by the SPUs. We wrote DMA operations in C language intrinsics,[12] which in most cases produce inline assembly instructions to specify commands to the MFC through the SPU channel interface. We measured elapsed time in the SPE using a special register, the SPU decrementer, which ticks every 40 ns (or 128 processor clock cycles).

PPE and SPE interactions were performed through mailboxes, input and output SPE registers that can send and receive messages in as little as 150 ns. We implemented a simple synchronization mechanism to start the SPUs in a coordinated way. SPUs notify the completion of benchmark phases through an atomic fetch-and-add operation on a main memory location that is polled infrequently and unintrusively by the PPE.

Basic DMA performance

The first step of our analysis measures the latency and bandwidth of simple blocking puts and gets, when the target is in main memory or in another SPE's local store. Table 1 breaks down the latency of a DMA operation into its components.

Figure 4a shows DMA operation latency for a range of sizes. In these results, transfers between two local stores were always performed between SPEs 0 and 1, but in all our experiments, we found no performance difference attributable to SPE location. The results show that puts to main memory and gets and puts to local store had a latency of only 91 ns for transfers of up to 512 bytes (four cache lines). There was little difference

between puts and gets to local store because local-store access latency was remarkably low—only 8 ns (about 24 processor clock cycles). Main memory gets required less than 100 ns to fetch information, which is remarkably fast.

Figure 4b presents the same results in terms of bandwidth achieved by each DMA operation. As we expected, the largest transfers achieved the highest bandwidth, which we measured as 22.5 Gbytes/s for gets and puts to local store and puts to main memory and 15 Gbytes/s for gets from main memory.

Next, we considered the impact of nonblocking DMA operations. In the Cell processor, each SPE can have up to 16 outstanding DMAs, for a total of 128 across the chip, allowing unprecedented levels of parallelism in on-chip communication. Applications that rely heavily on random scatter or gather accesses to main memory can take advantage of these communication features seamlessly. Our benchmarks use a batched communication model, in which the SPU issues a fixed number (the batch size) of DMAs before blocking for notification of request completion. By using a very large batch size (16,384 in our experiments), we effectively converted the benchmark to use a nonblocking communication model.

Figure 5 shows the results of these experiments, including aggregate latency and bandwidth for the set of DMAs in a batch, by batch size and data transfer size. The results show a form of performance continuity between blocking—the most constrained case—and nonblocking operations, with different degrees of freedom expressed by the increasing batch size. In accessing main memory and local storage, nonblocking puts achieved the asymptotic bandwidth of 25.6 Gbytes/s, determined by the EIB capacity at the endpoints, with 2-Kbyte DMAs (Figure 5b and 5f). Accessing local store, nonblocking puts achieved the optimal value with even smaller packets (Figure 5f).

Gets are also very efficient when accessing local memories, and the main memory latency penalty slightly affects them, as Figure 5c shows. Overall, even a limited amount of batching is very effective for intermediate DMA sizes, between 256 bytes and 4 Kbytes, with a factor of two or even three of bandwidth increase compared with the blocking case (for example, 256-byte DMAs in Figure 5h).

Table 1. DMA latency components for a clock frequency of 3.2 GHz.

Latency component	Cycles	Nanoseconds
DMA issue	10	3.125
DMA to EIB	30	9.375
List element fetch	10	3.125
Coherence protocol	100	31.25
Data transfer for inter-SPE put	140	43.75
Total	290	90.61

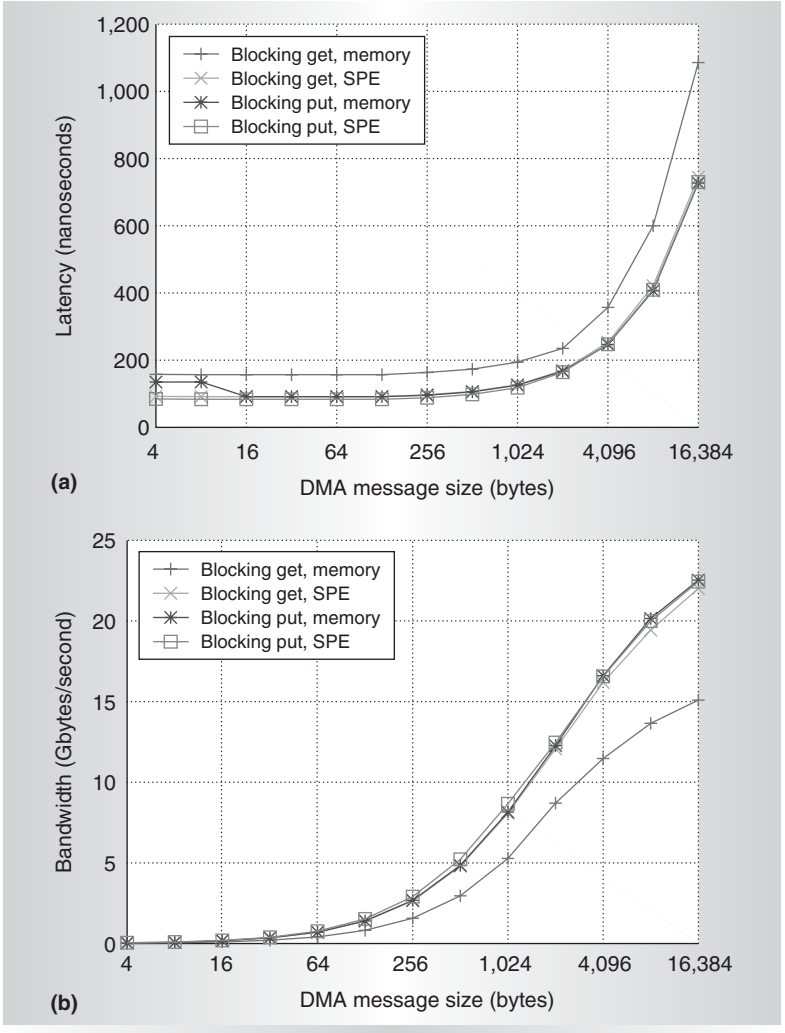


Figure 4. Latency (a) and bandwidth (b) as a function of DMA message size for blocking gets and puts in the absence of contention.

Collective DMA performance

Parallelization of scientific applications generates far more sophisticated collective

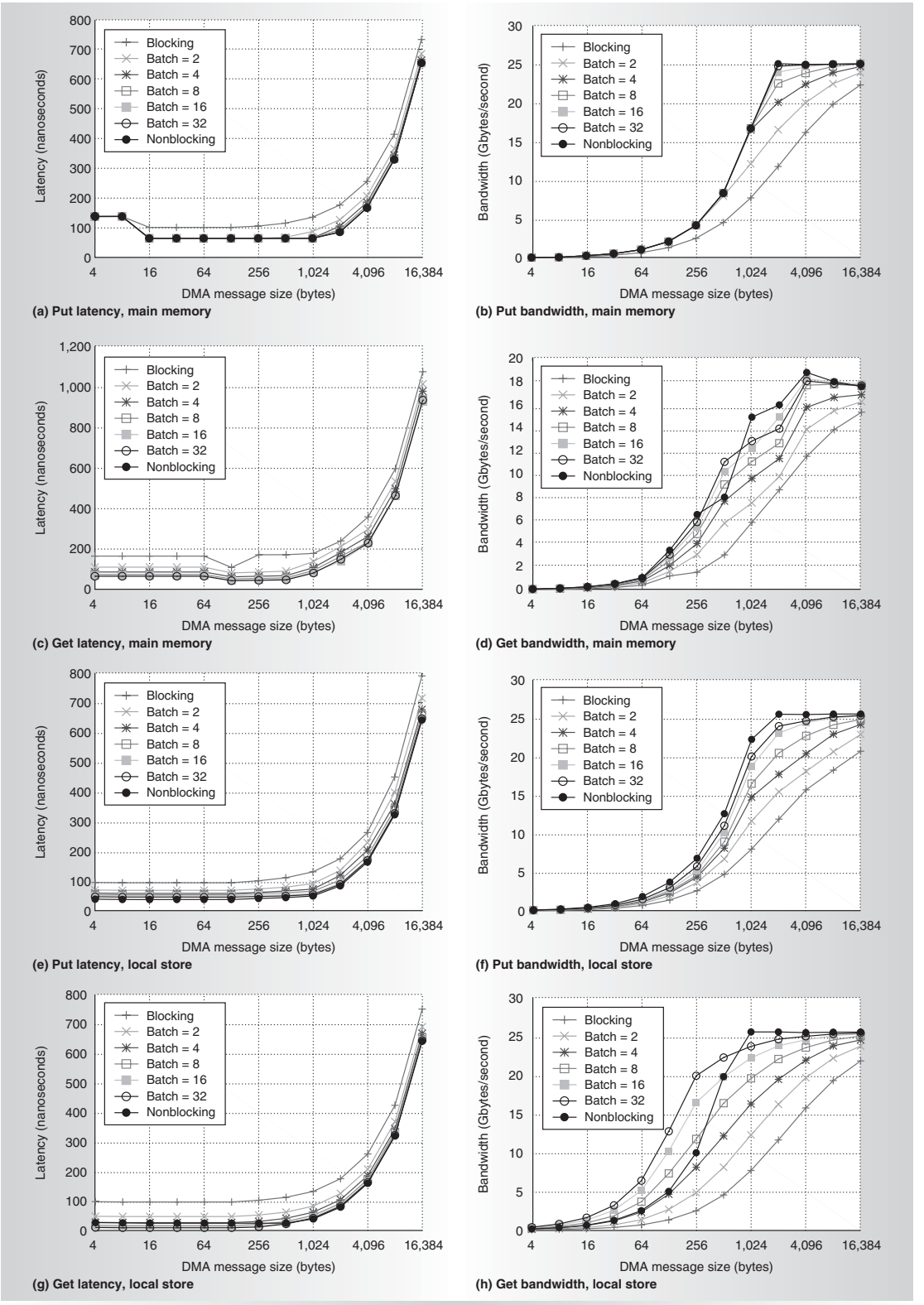


Figure 5. DMA performance dimensions: how latency and bandwidth are affected by the choice of DMA target (main memory or local store), DMA direction (put or get), and DMA synchronization (blocking after each DMA, blocking after a constant number of DMAs, ranging from two to 32, and nonblocking with a final fence).

communication patterns than the single pairwise DMAs discussed so far. We also analyzed how the system performs for several common patterns of collective communications under heavy load, in terms of both local performance—what each SPE achieves—and aggregate behavior—the performance level of all SPEs involved in the communication.

All results reported in this section are for nonblocking communication. Each SPE issues a sequence of 16,384-byte DMA commands. The aggregate bandwidth reported is the sum of the communication bandwidths reached by each SPE.

The first important case is the hot spot, in which many or, in the worst case, all SPEs are accessing main memory or a specific local store.[13] This is a very demanding pattern that exposes how the on-chip network and the communication protocols behave under stress. It is representative of the most straightforward code parallelizations, which distribute computation to a collection of threads that fetch data from main memory, perform the desired computation, and store the results without SPE interaction. Figure 6a shows that the Cell processor resolves hot spots in accesses to local storage optimally, reaching the asymptotic performance with two or more SPEs. (For the SPE hot spot tests, the number of SPEs includes the hot node; x SPEs include 1 hot SPE plus $x - 1$ communication partners.)

Counterintuitively, get commands outperform puts under load. In fact, with two or more SPEs, two or more get sources saturate the bandwidth either in main or local store. The put protocol, on the other hand, suffers from a minor performance degradation, approximately 1.5 Gbytes/s less than the optimal value.

The second case is collective communication patterns, in which all the SPES are both source and target of the communication. Figure 6b summarizes the performance aspects of the most common patterns that arise from typical parallel applications. In the two static patterns, complement and pairwise, each SPE executes a sequence of DMAs to a fixed target SPE. In the complement pattern, each SPE selects the target SPE by complementing the bit string that identifies the source. In the pairwise pattern, the SPEs are logically organized in pairs $< i, i + 1 >$, where i is an even number, and each SPE communicates with its partner. Note that SPEs with numerically consecutive numbers might not be physically adjacent on the Cell hardware layout.

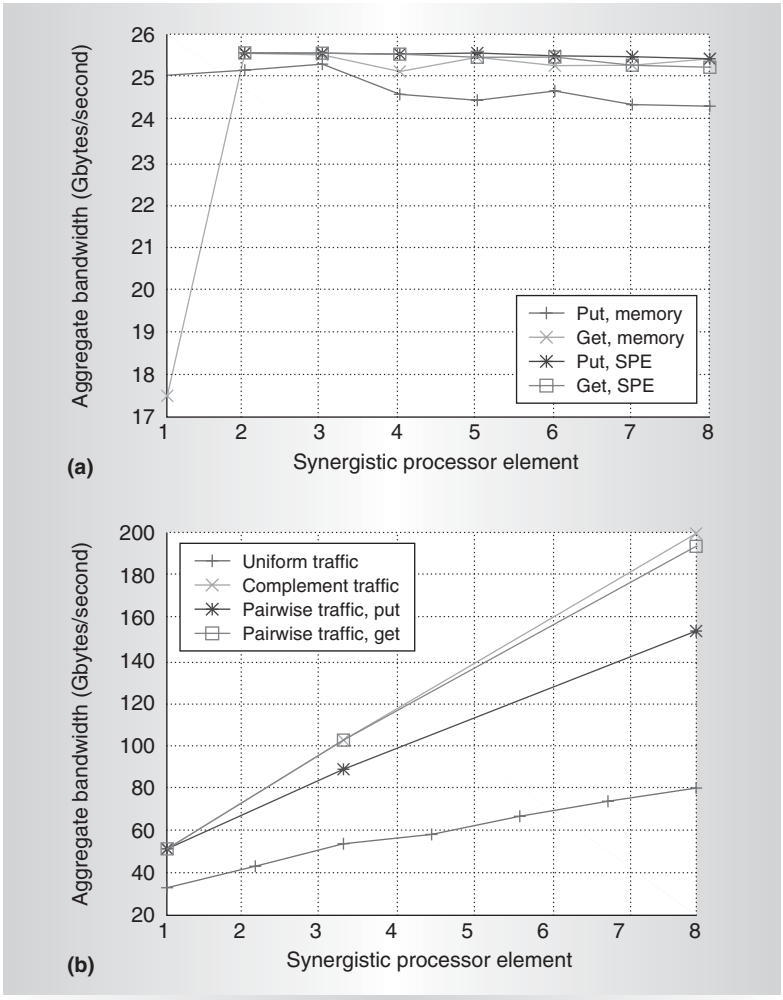


Figure 6. Aggregate communication performance: hot spots (a) and collective communication patterns (b).

The first static pattern, complement, is resolved optimally by the network, and can be mapped to the four rings with an aggregate performance slightly below 200 Gbytes/s (98 percent of aggregate peak bandwidth).

The direction of data transfer affects the pairwise pattern's performance. As the hot spot experiments show, gets have better contention resolution properties under heavy load, and Figure 6b further confirms this, showing a gap of 40 Gbytes/s in aggregate bandwidth between put- and get-based patterns.

The most difficult communication pattern, arguably the worst case for this type of on-chip network, is uniform traffic, in which each SPE

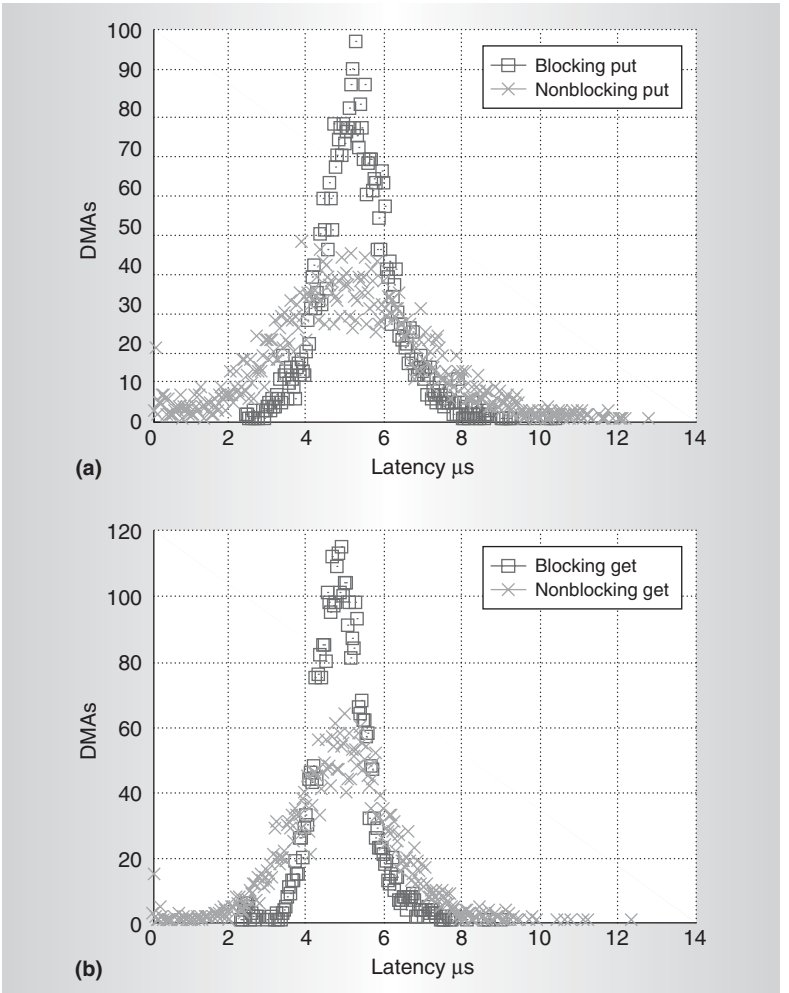


Figure 7. Latency distribution with a main-memory hot spot: puts (a) and gets (b).

randomly chooses DMA a target across SPEs' local memories. In this case, aggregate bandwidth is only 80 Gbytes/s. We also explored other static communication patterns and found results in the same aggregate-performance range.

Finally, Figure 7 shows the distribution of DMA latencies for all SPEs during the main-memory hot-spot pattern execution. One important result is that the distributions show no measurable differences across the SPEs, evidence of a fair and efficient algorithm for network resource allocation. The peak of both distributions shows a sevenfold latency increase, at about 5.6 µs. In both cases, the worst-case latency is only 13 µs, twice the average. This is another remarkable result, demonstrating that applications can rely on a responsive and fair network even with the most demanding traffic patterns.

Major obstacles in the traditional path to processor performance improvement have led chip manufacturers to consider multicore designs. These architectural solutions promise various power-performance and area-performance benefits. But designers must take care to ensure that these benefits are not lost because of the on-chip communication network's inadequate design. Overall, our experimental results demonstrate that the Cell processor's communications subsystem is well matched to the processor's computational capacity. The communications network provides the speed and bandwidth that applications need to exploit the processor's computational power. MICRO

References

1. T.M. Pinkston and J. Shin, "Trends toward On-Chip Networked Microsystems," *Int'l J. High Performance Computing and Networking*, vol. 3, no. 1, 2005, pp. 3-18.
2. J. Beecroft et al., "QsNet^{II}: Defining High-Performance Network Design," *IEEE Micro*, vol. 25, no. 4, July/Aug. 2005, pp. 34-47.
3. W.A. Wuld and S.A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM Sigarch Computer Architecture News*, vol. 23, no. 1, Mar. 1995, pp. 20-24.
4. U. Ghoshal and R. Schmidt, "Refrigeration Technologies for Sub-Ambient Temperature Operation of Computing Systems," *Proc. Int'l Solid-State Circuits Conf.* (ISSCC 2000), IEEE Press, 2000, pp. 216-217, 458.
5. R.D. Isaac, "The Future of CMOS Technology," *IBM J. Research and Development*, vol. 44, no. 3, May 2000, pp. 369-378.
6. V. Srinivasan et al., "Optimizing Pipelines for Power and Performance," *Proc. 35th Ann. Int'l Symp. Microarchitecture* (Micro 35), IEEE Press, 2002, pp. 333-344.
7. H.P. Hofstee, "Power Efficient Processor Architecture and the Cell Processor," *Proc. 11th Int'l Symp. High-Performance Computer Architecture* (HPCA-11), IEEE Press, 2005, pp. 258-262.
8. J.A. Kahle et al., "Introduction to the Cell Multiprocessor," *IBM J. Research and Development*, vol. 49, no. 4/5, 2005, pp. 589-604.
9. IBM Cell Broadband Engine Architecture 1.0,

http://www-128.ibm.com/developerworks/power/cell/downloads_doc.html.
10. IBM Full-System Simulator for the Cell Broadband Engine Processor, IBM AlphaWorks Project, http://www.alphaworks.ibm.com/tech/cellsystemsim.
11. J.L. Peterson et al., "Application of Full-System Simulation in Exploratory System Design and Development," *IBM J. Research and Development*, vol. 50, no. 2/3, 2006, pp. 321-332.
12. IBM SPU C/C++ Language Extensions 2.1, http://www-128.ibm.com/developerworks/power/cell/downloads_doc.html.
13. G.F. Pfister and V.A. Norton, "Hot Spot Contention and Combining in Multistage Interconnection Networks," *IEEE Trans. Computers*, vol. 34, no. 10, Oct. 1985, pp. 943-948.

Michael Kistler is a senior software engineer in the IBM Austin Research Laboratory. His research interests include parallel and cluster computing, fault tolerance, and full-system simulation of high-performance computing systems. Kistler has an MS in computer science from Syracuse University.

Michael Perrone is the manager of the IBM TJ Watson Research Center's Cell Solutions Department. His research interests include algorithmic optimization for the Cell processor, parallel computing, and statistical machine learning. Perrone has a PhD in physics from Brown University.

Fabrizio Petrini is a laboratory fellow in the Applied Computer Science Group of the Computational Sciences and Mathematics Division at Pacific Northwest National Laboratory. His research interests include various aspects of supercomputers, such as high-performance interconnection networks and network interfaces, multicore processors, job-scheduling algorithms, parallel architectures, operating systems, and parallel-programming languages. Petrini has a Laurea and a PhD in computer science from the University of Pisa, Italy.

Direct questions and comments about this article to Fabrizio Petrini, Applied Computer Science Group, MS K7-90, Pacific Northwest National Laboratory, Richland, WA 99352; fabrizio.petrini@pnl.gov.