Synergistic Processing in Cell's Multicore Architecture

Eight synergistic processor units enable the Cell Broadband Engine's breakthrough performance. The SPU architecture implements a novel, pervasively data-parallel architecture combining scalar and SIMD processing on a wide data path. A large number of SPUs per chip provide high thread-level parallelism.

Michael Gschwind
H. Peter Hofstee
Brian Flachs
Martin Hopkins
IBM

Yukio Watanabe
Toshiba

Takeshi Yamazaki
Sony Computer
Entertainment

•••••• When IBM, Sony, and Toshiba launched the Cell project[1] in 2000, the design goal was to improve performance an order of magnitude over that of desktop systems shipping in 2005. To meet that goal, designers had to optimize performance against area, power, volume, and cost, but clearly single-core designs offered diminishing returns on investment.[1-3] If increased efficiency was the overriding concern, legacy architectures, which typically incur a big overhead per data operation, would not suffice.

Thus, the design strategy was to exploit architecture innovation at all levels directed at increasing efficiency to deliver the most performance per area invested, reduce the area per core, and have more cores in a given chip area. In this way, the design would exploit application parallelism while supporting established application models and thereby ensure good programmability as well as programmer efficiency.

The result was the Cell Broadband Engine Architecture, which is based on heterogeneous chip multiprocessing. Its first implementation is the Cell Broadband Engine (Cell BE). The Cell BE supports scalar and single-instruction, multiple-data (SIMD) execution equally well and provides a high-performance multithreaded execution environment for all applications. The streamlined, data-processing-oriented architecture enabled a design with smaller cores and thus more cores on a chip.[4] This translates to improved performance for all programs with thread-level parallelism regardless of their ability to exploit data-level parallelism.

One of the key architecture features that enable the Cell BE's processing power is the *synergistic processor unit* (SPU)—a data-parallel processing engine aimed at providing parallelism at all abstraction levels. Data-parallel instructions support data-level parallelism, whereas having multiple SPUs on a chip supports thread-level parallelism.

The SPU architecture is based on *pervasively data parallel computing* (PDPC), the aim of which is to architect and exploit wide data paths throughout the system. The processor then performs both scalar and data-parallel SIMD execution on these wide data paths, eliminating the overhead from additional issues slots, separate pipelines, and the control complexity of separate scalar units. The processor also uses wide data paths to deliver instructions from memory to the execution units.
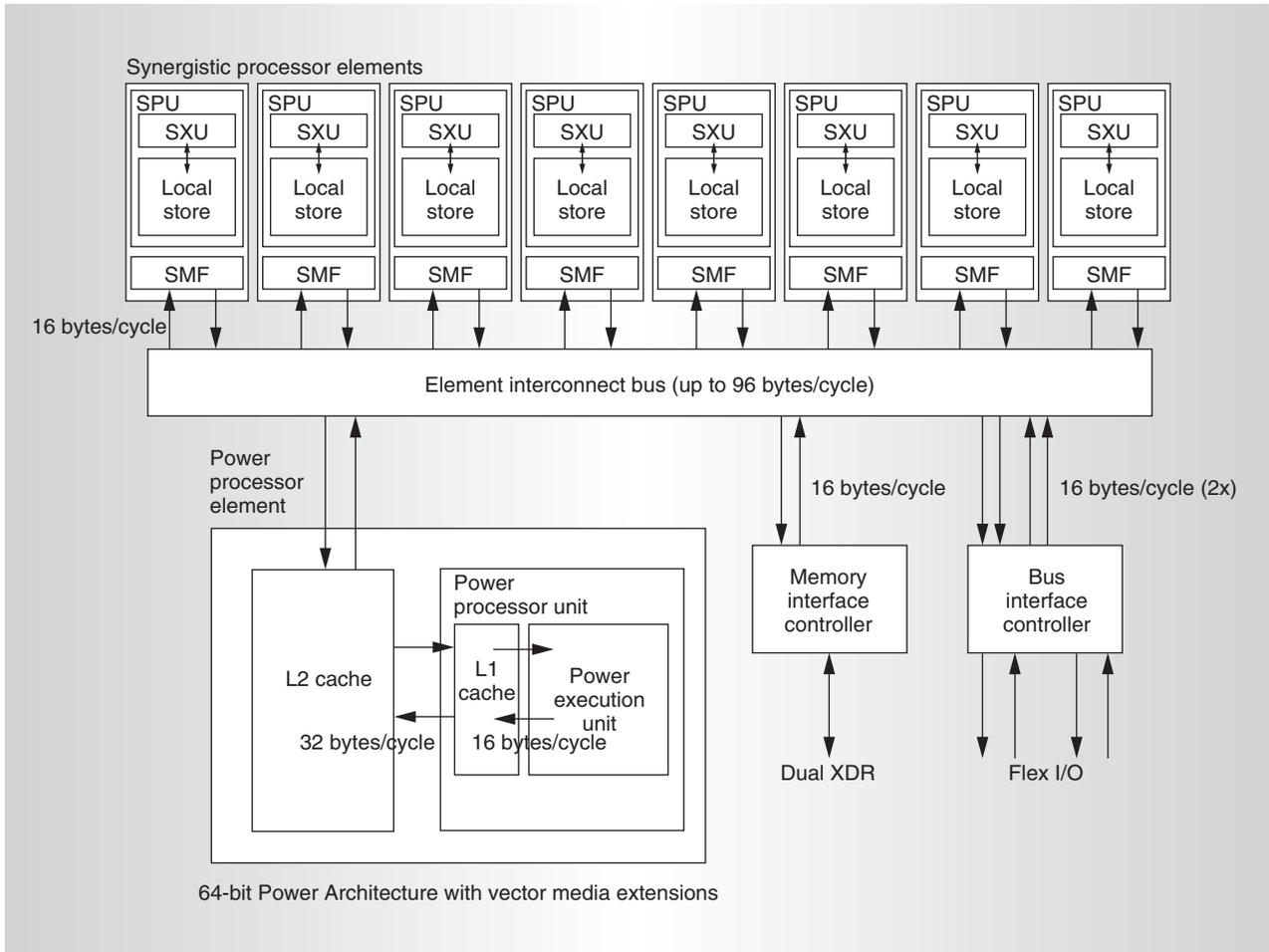
Figure 1. Cell system architecture. The Cell Broadband Engine Architecture integrates a Power processor element (PPE) and eight synergistic processor elements (SPEs) in a unified system architecture. The PPE is based on the 64-bit Power Architecture with vector media extensions and provides common system functions, while the SPEs perform data-intensive processing. The element interconnect bus connects the processor elements with a high-performance communication subsystem.

Overview of the Cell Broadband Engine architecture

As Figure 1 illustrates, the Cell BE implements a single-chip multiprocessor with nine processors operating on a shared, coherent system memory. The function of the processor elements is specialized into two types: the *Power processor element* (PPE) is optimized for control tasks and the eight *synergistic processor elements* (SPEs) provide an execution environment optimized for data processing. Figure 2 is a die photo of the Cell BE.

The design goals of the SPE and its architectural specification were to optimize for a low complexity, low area implementation.

The PPE is built on IBM's 64-bit Power Architecture with 128-bit vector media exten-

sions[5] and a two-level on-chip cache hierarchy. It is fully compliant with the 64-bit Power Architecture specification and can run 32-bit and 64-bit operating systems and applications.

The SPEs are independent processors, each running an independent application thread. The SPE design is optimized for computation-intensive applications. Each SPE includes a private local store for efficient instruction and data access, but also has full access to the coherent shared memory, including the memory-mapped I/O space.

Both types of processor cores share access to a common address space, which includes main memory, and address ranges corresponding to each SPE's local store, control registers, and I/O devices.
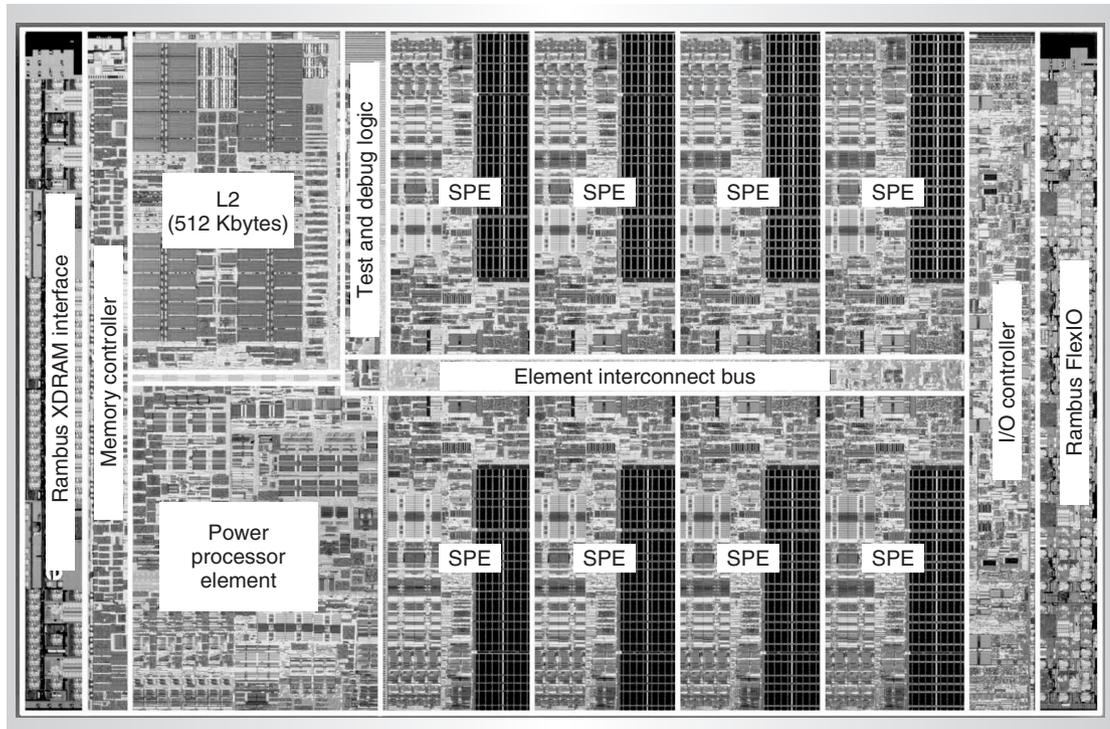
Figure 2. Die photo of the Cell Broadband Engine. The synergistic processor elements (including eight local stores) occupy only a fraction of the Power processor element area (including the L2 at top left) but deliver comparable computational performance.

Synergistic processing

The PPE and SPEs are highly integrated. The PPE provides common control functions, runs the operating system, and provides application control, while the SPEs provide the bulk of the application performance. The PPE and SPEs share address translation and virtual memory architecture, and provide support for virtualization and dynamic system partitioning. They also share system page tables and system functions such as interrupt presentation. Finally, they share data type formats and operation semantics to allow efficient data sharing among them.

Each SPE consists of the SPU and the synergistic memory flow (SMF) controller. The SMF controller moves data and performs synchronization in parallel to SPU processing and implements the interface to the element interconnect bus, which provides the Cell BE with a modular, scalable integration point.

Design drivers

For both the architecture and microarchitecture, our goal was not to build the highest single-core performance execution engine, but to deliver the most performance per area invested, reduce the area per core, and increase the number of cores (thread contexts) available in a given chip area. The design decisions described in this article exploit application characteristics for data-processing-intensive applications to improve efficiency. Other, more aggressive design decisions might have increased the per-core performance, but at the cost of larger cores and thus fewer cores in a given chip area.

Another design goal was to enable high-frequency implementations with modest pipeline depths and without deep sorting[6]— but without requiring the mechanisms that typically allow efficient instruction pipelining (register renaming, highly accurate branch predictors, and so on). Our solution was to reduce architectural complexity where feasible, subject to latencies from basic resource decisions such as the large register file (2 Kbytes) and large local store (256 Kbytes).

By providing critical system services (such as virtual memory support and system virtu-
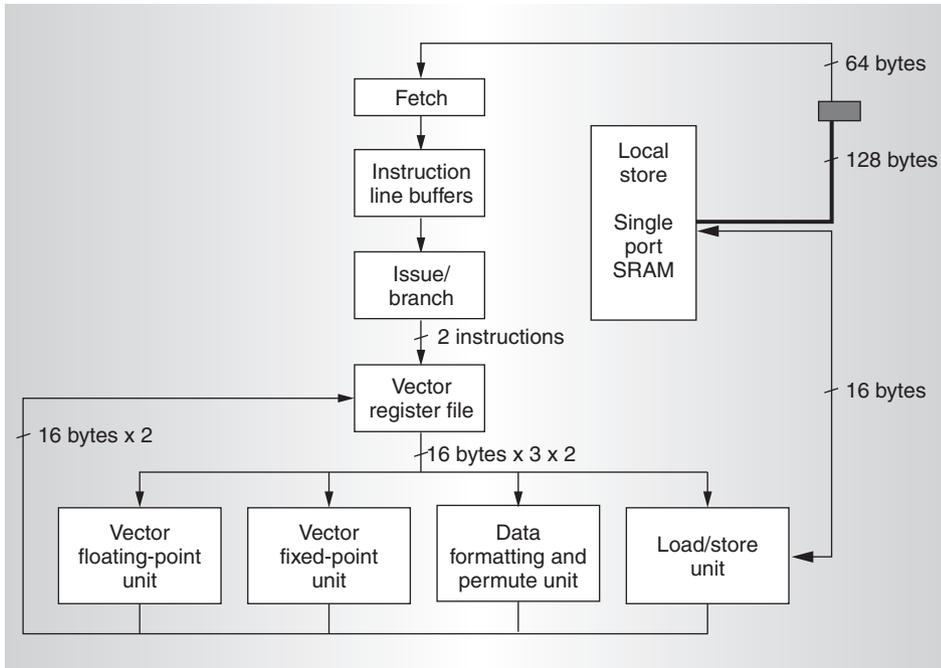
Figure 3. Synergistic processor unit. The SPU relies on statically scheduled instruction streams and embraces static compiler-based control decisions where feasible. The use of wide execution data paths reduces control overhead and increases the fraction of chip area and power devoted to data processing.

alization) in a Power Architecture core, the Cell BE avoids duplicating capabilities across all execution contexts, thereby using resources more efficiently. Providing two classes of cores also means that the design can optimize the PPE for control-dominated control code to dispatch high-volume data-processing tasks to cores optimized for data processing.

Synergistic processor unit

As Figure 3 shows, the SPU architecture promotes programmability by exploiting compiler techniques to target the data-parallel execution primitives. We essentially took from the lessons of reduced-instruction-set computing (RISC): The architecture provides fast, simple primitives, and the compiler uses these primitives to implement higher-level idioms. If the compiler could not target a set of functionality, we typically did not include it in the architecture.

To move decisions best performed at compile time into the compiler and thus reduce control complexity and power consumption, the architectural definition focused on exploiting the compiler to eliminate hardware com-

plexity. A simplified architectural specification also lets the hardware design optimize circuits for the common performance case, delivering lower latency and increasing area efficiency.

Pervasively data-parallel computing

Over the past decade, microprocessors have become powerful enough to tackle previously intractable tasks and cheap enough to use in a range of new applications. Meanwhile, the volumes of data to process have ballooned. This phenomenon is evident in everything from consumer entertainment, which is transitioning from analog to digital media, to supercomputing applications, which are starting to address previously unsolvable computing problems involving massive data volumes.

To address this shift from control function to data processing, we designed the SPU to exploit data-level parallelism through a SIMD architecture and the integration of scalar and SIMD execution. In addition to improving the efficiency of many vectorization transformations, this approach reduces the area and complexity overhead that scalar processing imposes. Any complexity reduction directly

translates into increased performance because it enables additional cores per given area.

For programs with even modest amounts of data-level parallelism, offering support for data-parallel operations provides a major advantage over transforming data-level parallelism into instruction-level parallelism. Legacy cores often take the latter approach, which requires processing and tracking the increased number of instructions and often yields significant penalties because parallelism must be rediscovered in instructions using area- and power-intensive control logic.

Data alignment for scalar and vector processing

In existing architectures, only limited, subword arithmetic byte and halfword data paths could share logic between scalar and vector processing; more general vector processing required a separate data-parallel data path. To streamline the design, we departed from this practice. The SPU has no separate support for scalar processing.

To be consistent with the data-parallel focus, we also optimized the memory interface for aligned quadword access, thus eliminating an elaborate alignment network typically associated with scalar data access. This design decision reduces control complexity and eliminates several latency stages from the critical memory access path, similar to the original MIPS-X and Alpha architectural specifications. It also reduces overall latency when dynamic data alignment is not required. When it is required, either to access unaligned vectors or to extract scalar data not aligned on a 128-bit boundary, the compiler can insert vector shuffle or rotate instructions to align and extract data. The cumulative delay of this quadword data access and a separate alignment instruction corresponds to the delay of a memory operation implementing data alignment.

The decision to use a software-controlled data-alignment approach is synergistic with the integration of scalar and vector processing and the emphasis on processing through wide data paths. From a workload perspective, the quadword memory interface supports data-parallel (short vector) processing, which means that the Cell BE can perform array accesses to successive elements without repeated extraction by operating on multiple data elements in parallel. From a system architecture perspective, there are no complications with device driver code, such as those experienced in the Alpha environment, because each SPU uses the SMF controller and its direct memory access facility for I/O accesses.

Following the PDPC concept, the SPU architecture does not include a separate scalar register file. This would complicate data routing for source operands and computational results, require additional routing and multiplexers (with their associated latency), and represent additional loads on result buses. The SPU stores all scalar data in a unified, 128-entry, 128-bit-wide scalar/vector register file that feeds directly into the processing functions performed in wide SIMD execution data paths. Using a unified scalar/SIMD register also simplifies performing scalar data extraction and insertion and data sharing between scalar and vector data for parallelizing compiler optimizations.[7,8]

The unified register file also stores data of all types, which means that a single register file stores integer values, single- and double-precision floating-point values, Boolean values, and addresses. The register file can provide a single quadword element, two 64-bit doubleword elements, four 32-bit word elements, eight 16-bit halfword elements, 16-byte elements, or a vector of 128 single-bit elements.

The program can use all 128 entries to store data values, and the register file is fully symmetric from an architecture perspective. No registers are hardwired to specific values, which would require expensive special handling during instruction decode and register file access and in bypass and forwarding logic. All instructions can reference any of the 128 registers—that is, no instructions must use an instruction-specific implicit register or a register file subset.

The aim of these design decisions is to increase compiler efficiency in register allocation. Using a single unified register file lets compiler and application programmers allocate resources according to specific application needs, improving programmability and resource efficiency.

Scalar layering

We call the execution of scalar operations on wide SIMD data paths *scalar layering*. It has two
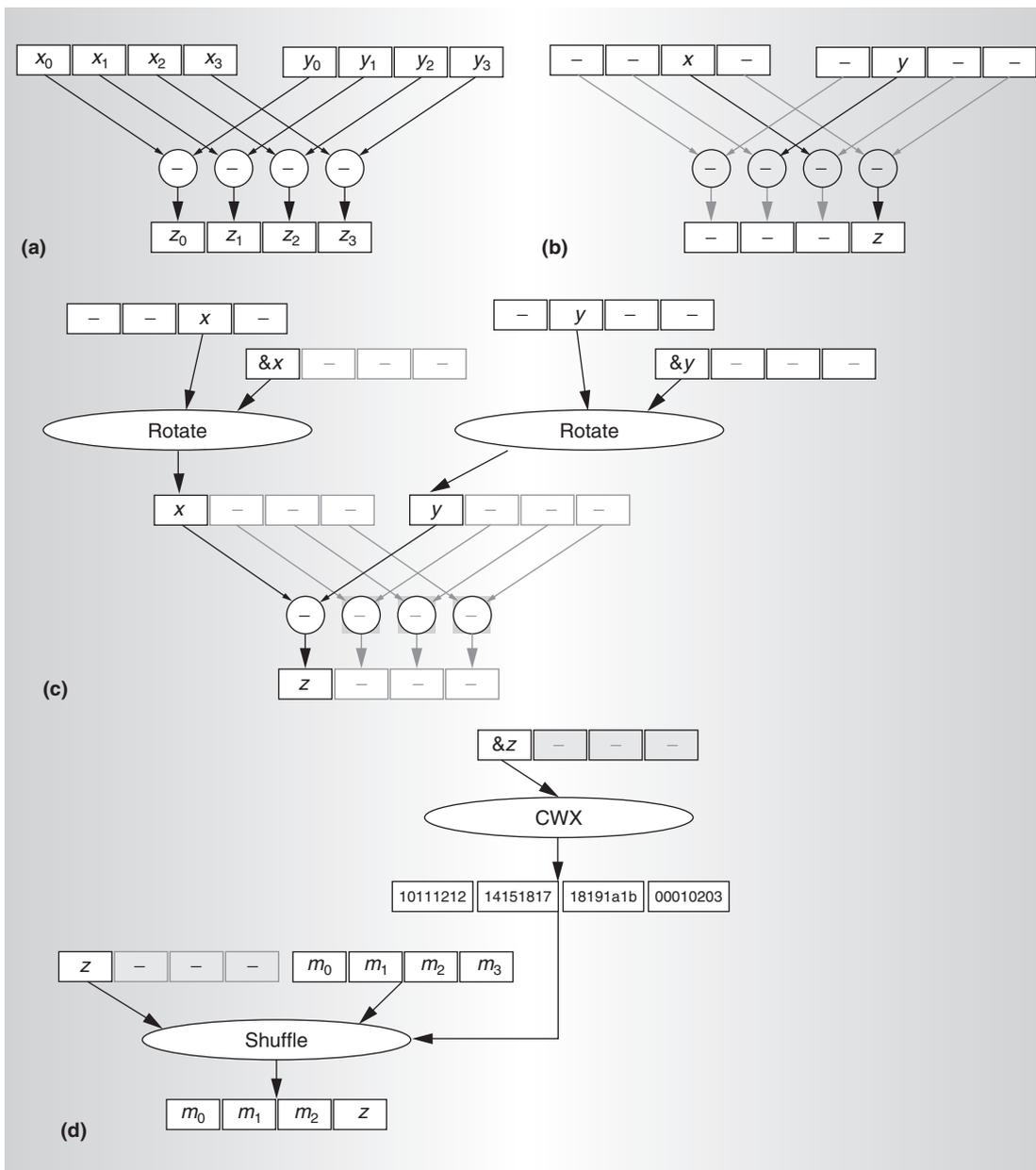
Figure 4. How scalar layering works. Scalar layering aligns scalar data under compiler control. (a) SIMD operation, (b) alignment mismatch of scalar elements in vector registers without data alignment, (c) operations with scalar layering, compiler-managed scalar extraction, and data alignment, and (d) subvector write using optimized read-modify-write sequence.

aspects: scalar operations mapping onto the data-parallel execution engines and data management to align, extract, and insert data on memory accesses using the memory interface.

To illustrate how scalar layering works, consider the operation of SIMD data-parallel execution pipelines as described earlier on a four-element vector consisting of one word

each. Figure 4a illustrates how a processor executes a SIMD instruction by performing the same operation—in parallel—on each element. In the example, the SIMD instruction sources two vector registers containing elements x_0, x_1, x_2, and x_3 and y_0, y_1, y_2, and y_3, respectively, and yields four results: $z_0 = x_0 - y_0$; $z_1 = x_1 - y_1$; $z_2 = x_2 - y_2$; and $z_3 = x_3 - y_3$ in

a vector register allocated to result z_0, z_1, z_2, and z_3. Figure 4b shows that SIMD data-parallel operations cannot readily be used for operations on scalar elements with arbitrary alignment loaded into a vector register using the quadword load operations. Instead, data has to be aligned to the same slot.

Figure 4c shows the compilation of scalar code to execute on a SIMD engine. The example is based on performing the computation in the leftmost slot, but a compiler—or programmer—can align scalar operands to any common slot to perform operations. On the basis of the alignment that the scalar word address specifies, rotate instructions align scalar data in the selected vector slot from the quadword that memory access retrieves (rotate instructions obtain their shift count from the vector's leftmost slot). Once the SPU aligns memory operands to a common slot, the SPU will perform all computations across the entire SIMD vector.

Figure 4d illustrates the use of a read-modify-write sequence to store scalar data via the quadword-oriented storage interface. To process scalar operations, the SPU uses a compiler-generated layering sequence for memory accesses when it must merge scalar data into memory.

The SPU inserts a scalar element in a quadword by using the shuffle instruction to route bytes of data from the two input registers. To implement the read-modify-write sequence, the SPU also supports a "generate controls for insertion" instruction, which generates a control word to steer the shuffle instruction to insert a byte or halfword or word element into a position the memory address specifies.

All streaming accesses for data-parallel SIMD processing—except the first and last accesses, which could represent partial accesses for improperly aligned streams—can exploit accesses without the latency, power, and area penalty of implementing an additional merge network in the memory store path. Streaming accesses predominate in a PDPC architecture. By making data alignment distinct from memory access with a separate instruction, the compiler can attempt to optimize data layout to reduce this cost. Aligned quadword loads do not incur the latency penalty for extracting and aligning subwords, because the memory interface is optimized to transfer and store entire quadwords in both memory and registers. This cost avoidance directly benefits SIMD vector operations.

Scalar code sequences also do not incur the extraction and alignment penalty implicit in memory accesses. Our solution was to make a large register file available so that the SPU can access many variables that would otherwise spill into memory directly from the register file. Reducing the number of opcode points (and hence the opcode field size) assigned to different load and store variants makes it easier to encode multiple 7-bit register specifiers in a 32-bit instruction word.

In generating code for the SPU, a compiler can allocate scalar values that must be spilled during register allocation to a full quadword spill area to spill and reload the entire register. Additionally, it can tailor function call and return sequences to start and end spilling at aligned boundaries. In this way, these sequences can efficiently pack scalar call site spills and reduce memory traffic and instruction count. Exploiting statically known alignment and selecting an optimized slot within the vector for intermediate computations are still other ways to achieve compiler-based performance improvements.

Our decision to use data-parallel execution paths to implement scalar processing simplifies the control logic to dispatch instructions by reducing the number of execution units to which the SPU can issue the instruction. This also results in reduced fan-out and wire load on the issue logic, less dependence checking and bypass logic, and fewer register file ports.

The PDPC architecture also facilitates the sharing of scalar and data parallel results. This in turn makes SIMD vectorization more efficient because of the lower data synchronization and movement cost.

Optimizing scalar processing

Many instructions require scalar operands, but in an architecture with only vector registers, it is not sufficient to specify a register containing a vector of multiple scalar values. To resolve scalar operand references, the SPU architecture convention is to locate these operands in the vector's "preferred slot," which as Figure 5 shows, corresponds to the leftmost word element slot, consisting of bytes b_0 to b_3. Instructions using the preferred slot concept
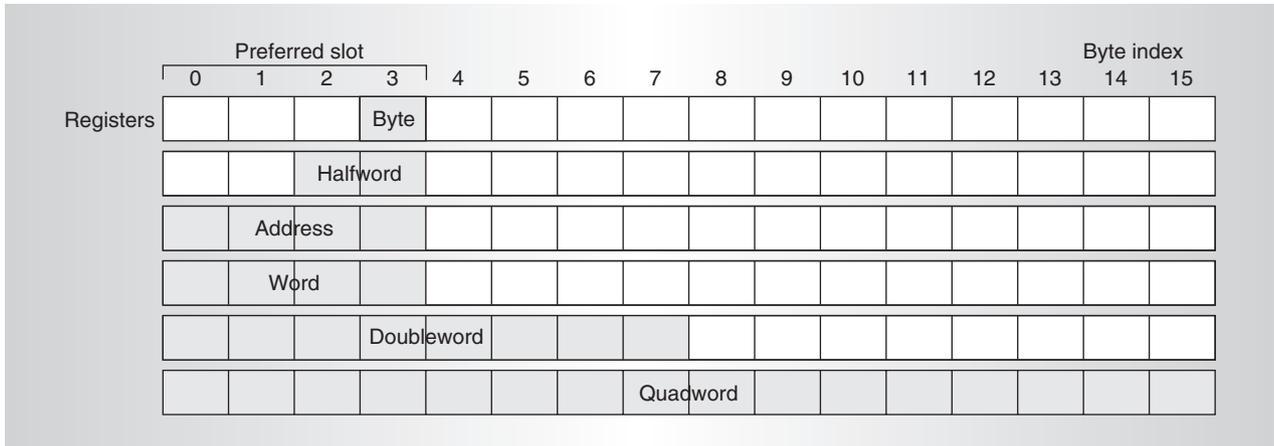
Figure 5. Data placement of scalar values in a vector register using the preferred slot. SPU architecture convention allocates scalar values in the leftmost slot for instruction operands and function call interfaces, but the compiler can optimize code to perform scalar computation in any vector element slot.

include shift and rotate instructions operating across an entire quadword to specify the shift amount, memory load and store instructions that require an address, and branch instructions that use the preferred slot for branch conditions (for conditional branches) and branch addresses (for register-indirect branches). Branch and link instructions also use the preferred slot to deposit the function return address in the return address register, which the Cell application binary interface (ABI) allocates to vector register 0.

The preferred slot is the expected location for scalar parameters to SPU instructions, but scalar computation can occur in any slot. The preferred slot also serves as a software abstraction in the ABI to identify the location of scalar parameters on function calls and as function return values. Interprocedural register allocation can choose alternative locations to pass scalar values across function call boundaries.

Initially, the SPU architecture specification called for an indicator bit in the instruction encoding for all vector instructions to indicate their use for scalar operations. This meant that the processor would compute only results in the preferred slot range and essentially deenergized up to 75 percent of the data path. However, a test chip showed that this optimization offered only limited power reduction because of the focus on data-parallel processing (most instructions exploit multiple parallel execution lanes of a SIMD data path) as well as the increased control complexity of supporting different instruction types. When operands of different widths are in the data path, control decision complexity increases. For example, bypassing becomes nonuniform when bypassing between data of different widths. This in turn requires multiple independent bypass networks and the ability to handle boundary conditions, such as injecting default data values. Thus, if enough instructions in the mix are wide, the power savings potential drops, overshadowed by the increased power spent in more complex control and data routing.

For that reason, the current SPU architecture specification contains only vector instruction forms, and the scalar nature of an instruction can be inferred only from how the compiler uses that instruction, not by any form. The compiler selects a slot position in a vector in which to perform intermediate computations and from which to retrieve the result. The hardware is completely oblivious of this use and always performs the specified operation across all slots.

Removing explicit scalar indication, however, is precisely what frees the software to perform scalar operations in any element slots of a vector. The compiler can optimize alignment handling and eliminate previously compulsory scalar data alignment to the preferred slot. Unifying instruction encoding in this way makes more opcode bits available to encode operations with up to four distinct operands from a 128-entry register file.
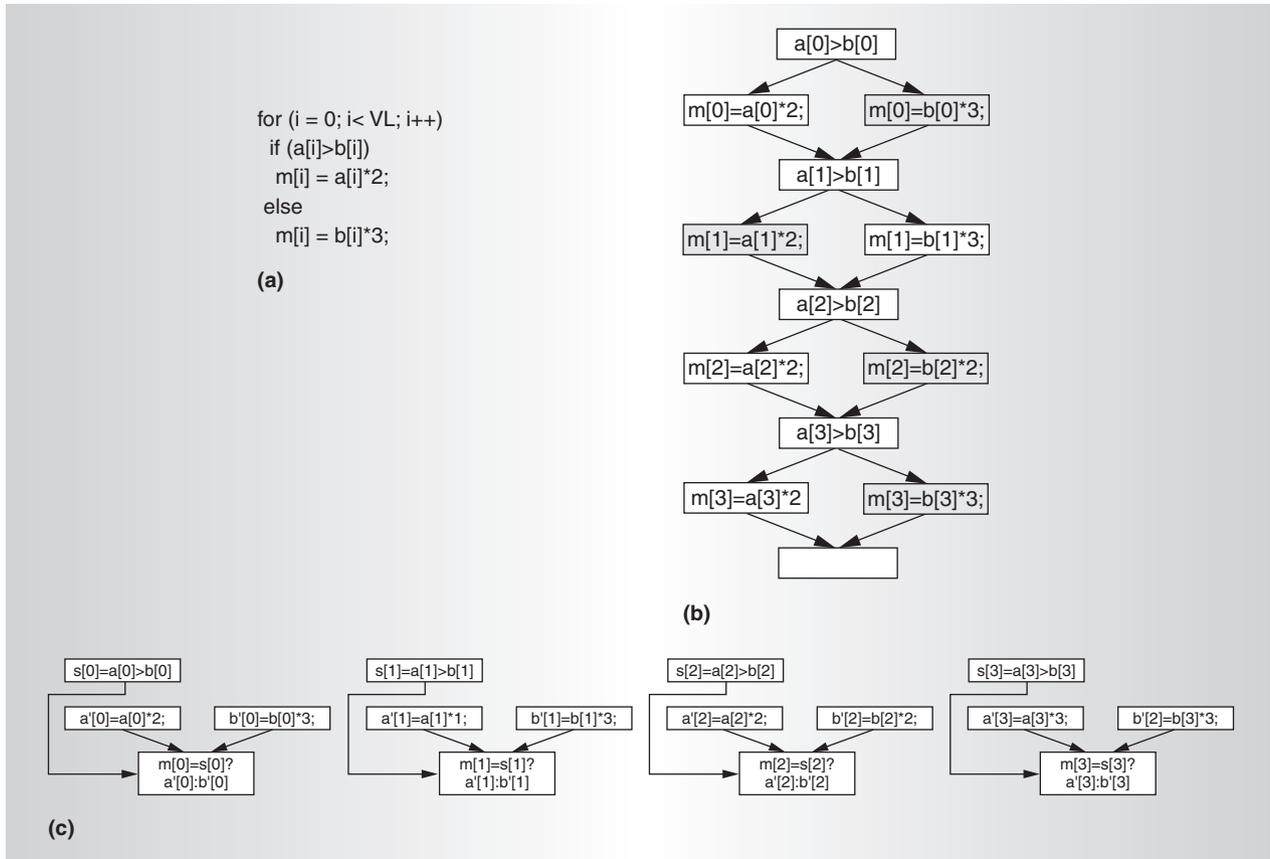
Figure 6. The use of data-parallel select to exploit data parallelism. (a) Conditional operations are integrated into SIMD-based computation. (b) Using traditional code generation techniques, the source code is turned into a sequence of test and conditional branch instructions for each vector element. High branch misprediction rates of data-dependent branches and data conversion between vector and scalar representations incur long schedules. (c) Exploiting data-parallel conditional execution with data-parallel select allows the processing of conditional operations concurrently on multiple vector elements. In addition to exploiting data parallelism, data-parallel select purges hard-to-predict data-dependent branches from the instruction mix.

Data-parallel conditional execution

Many legacy architectures that focus on scalar computation emphasize the use of conditional test and branch to select from possible data sources. Instead, following the focus on PDPC, we made data-parallel select the preferred method for implementing conditional computation. The data-parallel select instruction takes two data inputs and a control input (all stored in the unified register file) and independently selects one of the two data inputs for each vector slot under the control of the select control input. Using data-parallel select to compute the result of conditional program flow integrates conditional operations into SIMD-based computation by eliminating the need to convert between scalar and vector representation. The resulting vec-

torized code thus contains conditional expressions, which in turn lets the SPU execute conditional execution sequences in parallel.

As Figure 6 shows, to use conditional branch operations, the compiler must translate a simple element-wise data selection into a sequence of scalar conditional tests, each followed by a data-dependent branch. In addition to the sequential schedule, each individual branch is data dependent and many branches are prone to misprediction by even the most sophisticated dynamic prediction algorithm. This results in a long latency control-flow dominated instruction schedule as shown in Figure 6b, exacerbated by a significant penalty for each mispredicted branch. The control-dominated test-and-branch sequence must be embedded between code to

unpack a vector into a sequence of scalar values and followed by code to reassemble the scalar result into a vector.

The preferred method for conditional execution on the SPU is to exploit data parallelism and implement conditional execution with a short sequence of data-parallel SIMD instructions, as Figure 6c shows. The data-parallel select sequence replaces the lengthy test-and-branch sequence with four instructions (two multiplies, one compare, and a data-parallel select instruction) operating on a vector of four elements. By using data-parallel if-conversion to execute both paths of a conditional assignment, each path can execute on the full vector—effectively reducing the number of executed blocks from once for each vector element (using scalar branch-based code) to once for each execution path.

This emphasis on parallel execution offers significant advantages over the control-dominated compare-and-branch sequence. If-conversion creates opportunities for exploiting transformations that enhance instruction-level parallelism, like software pipelining. Such transformations become easier to perform with a simple dataflow graph.

In addition to these familiar benefits of if-conversion, data-parallel select is a basis for exploiting data-level parallelism. Historically, predicated architectures have suffered from unbalanced then-else paths, where one execution path is inordinately longer than the other, or the distribution between execution probabilities is widely skewed. In a data-parallel environment, these trade-offs are more favorable for data-parallel select.

In applying predication to scalar code, the number of executed instructions corresponds to the sum of the instructions executed along either execution path. To offset this increased instruction count, scalar predication reduces branch prediction penalties and improves code scheduling.

In applying predication to SIMD execution, data-parallel select offers an aggregate path length advantage by exploiting data-level parallel SIMD processing in addition to the traditional advantages of predication. This SIMD path length advantage offsets the potential cost of misbalanced then-else paths. Predication applied to SIMD execution offers to reduce path length to the aggregate path

length of the sum of instructions along one instance of the short path and one instance of the long path, compared to the sum of instructions on $p * w$ short paths, and $(1 - p) * w$ long paths, where p is the probability of executing a short path for a given execution, and w is vector width. This makes data-parallel select attractive except for very skewed probabilities or highly nonuniform distributions within these probabilities.

Essentially, data-parallel select turns a data-driven branch sequence prone to high misprediction rates into a dataflow operation. It removes conditions that are hard to predict statically from the instruction mix, thus skewing the mix toward easier-to-predict branches.[9] Increasing sequential control flow also increases opportunities for sequential fetch and reinforces the advantages of the static scheduling architecture.

The data-parallel select architecture integrates with the data-parallel compare architecture. All compare operations produce a data-width-specific control word to feed as control input into the data-parallel select operation. In addition, the result in the leftmost element slot (preferred slot) is potential input for a conditional branch instruction.

The SPU implements only two types of compare operations for each data type: one for equality and one for ordering. Compilers and assembly language programmers can derive all other conditions by inverting the order of operands (for compare and select operations) and by testing the condition or the inverted condition (for branch instructions).

Ensuring data fidelity

Many existing SIMD instruction sets emphasize processing throughput over data quality by using short data types and saturating arithmetic. In contrast, the SPU architecture's data type and operation repertoire emphasize halfword and word data types with traditional two's complement integer arithmetic and floating-point formats. The repertoire also provides a carefully selected set of byte operations to support efficient implementation of video compression and cryptography algorithms.

To avoid the data fidelity loss associated with saturating roundoff, the SPU does not support integer-saturating arithmetic, usually

attractive for low-cost media rendering devices. Instead, the SPU programming model extends narrow data types and avoids repeated roundoff during computation. Pack-and-saturate operations pack the final results to reduce the memory footprint.

Applications that require saturating arithmetic exploit floating-point operations, which are naturally saturating and offer a wide dynamic range. The SPU implements single-precision floating-point arithmetic optimized for graphics with an IEEE-compatible data format. The graphics-optimized format eliminates traps and exceptions associated with IEEE arithmetic and substitutes appropriate default values to avoid disrupting real-time media processing from overflow or underflow exception conditions.

The SPU also implements IEEE-compatible double-precision floating point arithmetic with full support for IEEE-compatible rounding, NaN-handling, overflow and underflow indication, and exception conditions.

Deterministic data delivery

Coherence traffic, cache-miss handling, and latencies from variable memory accesses negatively affect compiler scheduling. To avoid these costs, the SPE includes support for a high-performance local store that applications can use in conjunction with data privatization.

All memory operations that the SPU executes refer to the address space of this local store. Each SPU uses a private local store, which provides a second level of data storage beyond the large register file. Current SPU implementations support a local store of 256 Kbytes, with architectural support for up to a 4-Gbyte address range. A local store limit register (LSLR) lets designers limit the addressable memory range to promote compatibility among generations of implementations with possibly different local store sizes.

From a hardware perspective, a local store allows denser implementation than cache memory by eliminating tags and associated cache maintenance state as well as cache control logic. Eliminating cache coherence traffic also reduces the amount of necessary snoop traffic, which makes the element interconnect bus more efficient.

Figure 2 shows the difference between the 256-Kbyte SPU local store and a traditional 512-Kbyte L2 cache with support logic. Elim-

inating complex and time-critical cache-miss handling lets the SPU deliver lower latency external data access. From a hardware-software codesign perspective, replacing a sequence of cache-miss-induced single cache line requests (typically 32 to 128 bytes) with a block data transfer request of up to 16 Kbytes increases the efficiency in using the memory interface, since applications can fetch data ahead of use, on the basis of application behavior. The SMF controller implements the SPE's interface to the element interconnect bus for data transfer and synchronization. Because the SMF controller is an independent processing unit optimized for data transfer, each SPE can perform data processing and data transfer using software pipelining and double buffering of data transfer requests in parallel, making more parallelism available to application programmers.[10]

From a software perspective, the local store offers low and deterministic access latency, which improves the effectiveness of many compiler-based optimizations to hide latency, such as instruction scheduling, loop unrolling, and software pipelining.[7]

Both memory read and write operations return a single aligned quadword by truncating the low-order four address bits. When the application must load data that crosses a quadword boundary, the SPU uses the shuffle byte operation to perform a sequence involving two load operations and a data merge operation. The Cell ABI requires alignment of scalar values on their natural alignment boundary (not crossing a quadword boundary), but vector words can cross the quadword boundary because of data access patterns. Thus, even if the underlying array is aligned at a 128-bit boundary, selecting subarrays could yield unaligned data values (accessing the second element of a 128-bit aligned array, for example, yields an unaligned vector address).

Unaligned load operations might seem to provide relief in this context, but all implementation options that support unaligned access have a substantial cost. One implementation option is to preallocate bandwidth to perform two accesses at instruction issue time, thereby reducing the available bandwidth by 2 times even if no unaligned accesses are required. Another option is to optimistically assume aligned access, perform a pipeline flush, and
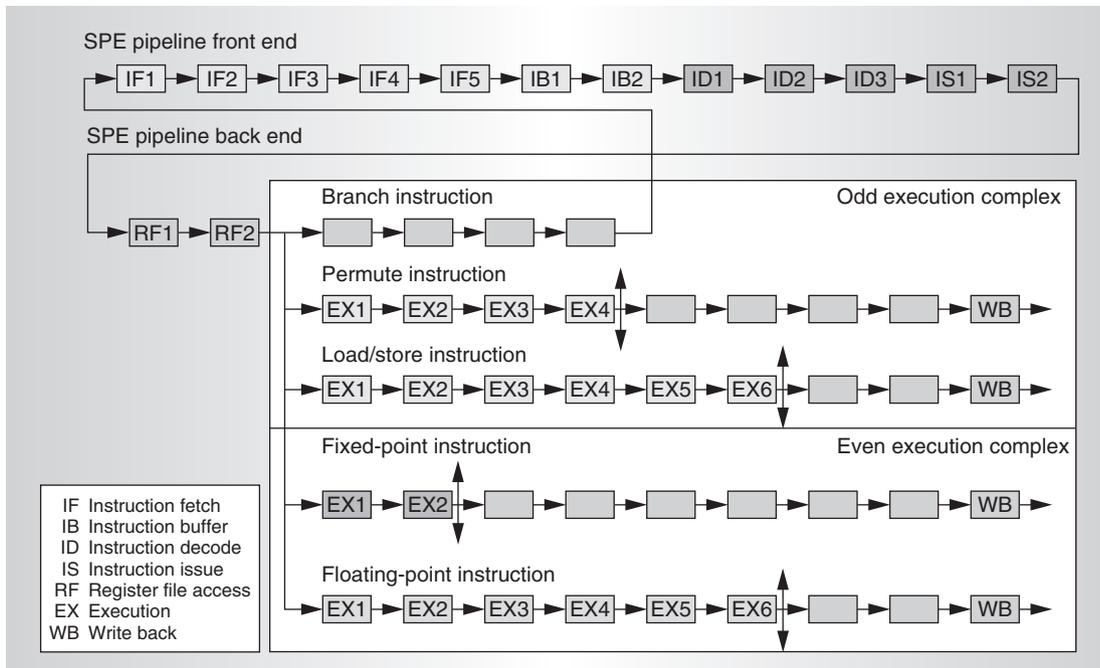
Figure 7. Microarchitecture for the SPE pipeline. The SPE architecture is bundle-oriented, supporting the delivery of up to two instructions per cycle to the data-parallel back end. The architecture supports very high frequency operation with comparatively modest pipeline depths by reducing architectural complexity.

reexecute a recovery sequence in the event of unaligned accesses. However, every unaligned access incurs a substantial performance penalty. Because both these solutions have high penalties, we opted to use a compiler-aided alignment policy. When the compiler cannot determine alignment statically, the compiler generates explicit dual-load and data-merge sequences for vector accesses. Most vector accesses are part of longer loops, so the actual throughput of load operations approaches one load per quadword loaded for common unit stride streams, since two iterations can share each load operation as an iteration-carried dependence. Compilation techniques to exploit this feature are available in the literature.[7]

The local store also serves as storage for program instructions that the SPU will execute (see Figure 2). The SPU fetches instructions with 128-byte accesses from a wide fetch port, delivering 32 instructions per access. It stores instructions in instruction line buffers and delivers them to the execution pipelines. Exploiting wide accesses for both instruction and data accesses decreases the necessary accesses and improves power efficiency. Instruction and data accesses and the SMF

controller share a single SRAM port, which improves memory density and reduces the latency for local store access.

Statically scheduled instruction-level parallelism

In the Cell BE, the SPU front end implements statically scheduled instruction fetch to reduce the cost of dynamic instruction scheduling hardware. However, the SPU architecture is not limited to implementations using static scheduling.

The SPU architecture is bundle-oriented and supports the delivery of up to two instructions per cycle to the data parallel back end. All instructions respect their dependencies to ensure sequential program semantics for future architectural compatibility and to ensure good code density.

The use of compiler-generated bundling simplifies instruction routing logic. By relying on compile-time scheduling to optimize instruction layout and encode instruction streams in bundles, the SPU eliminates much of the overhead of dynamic scheduling. As Figure 7 shows, instruction execution resources fall into one of two execution complexes: odd
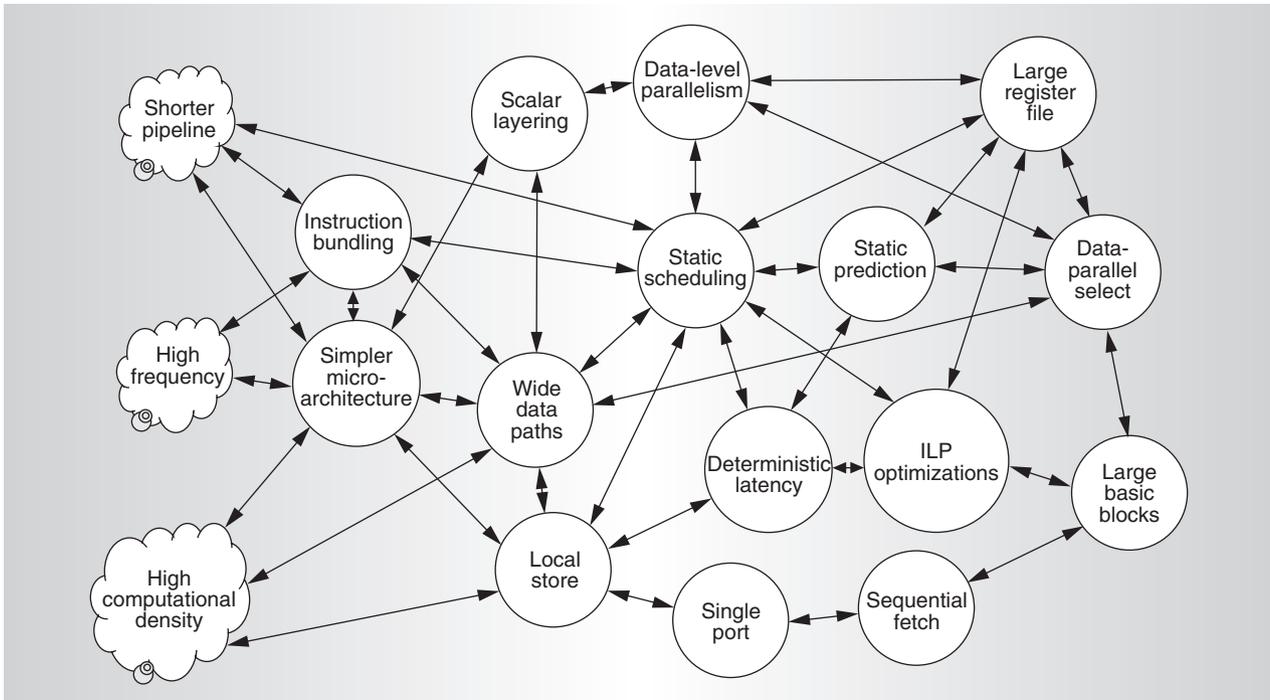
Figure 8. How design goals and decisions have led to synergy across the architecture. To the left are the main three design goals, while at the far right are design decisions. Interdependence increases in the center concepts.

or even. Instruction bundles can dual-issue instructions if a bundle is allocated at an even instruction address (an address that is a multiple of 8 bytes), and the two bundled instructions have no dependencies. In a dual-issued bundle, the SPU executes the first instruction in the even execution complex, and a second instruction in the odd execution complex. As the figure shows, the even execution complex consists of fixed and floating-point execution pipelines; the odd execution complex includes memory access and data formatting pipelines, as well as branches and channel instructions for communicating with the SMF controller.

We also applied the idea of statically scheduled instruction execution to the branch prediction architecture, which implements static branch prediction with a prepare-to-branch instruction. The compiler inserts this branch hint instruction to predict the target of branch instructions and initiate instruction prefetch from the predicted branch target address. The prepare-to-branch instruction accepts two addresses, a trigger address and a target address, and fetches instructions from the specified target address into a branch target buffer. When instruction fetch reaches the trigger address, the instruction stream continues execution with instructions from the target buffer to avoid a branch delay penalty. Both mispredicted and nonhinted taken branches incur a misprediction penalty.

In addition to the static branch prediction, the architecture supports compiler-controlled sequential instruction fetch primitives to avoid instruction starvation during bursts of high-priority data memory accesses that might otherwise preempt the instruction fetches. Instruction fetches return a full 128-byte line per access.

Simplicity and synergy

We defined the SPU architecture from an intense focus on simplicity and synergy. Our overarching goal was to avoid inefficient and expensive superpipelining in favor of optimizing for the common performance case. The compiler aids in layering traditional hardware functions in software to streamline the architecture further and eliminate nonessential functionality, thereby shortening latencies for common-case operations.

The synergy comes from mutually reinforcing design decisions, as Figure 8 illustrates.

At the left are the main design goals: high computational density, high frequency, and shorter pipelines. We believe that we have successfully met these goals and avoided the performance degradation often found in high-frequency designs.[3]

To the right are some of the design decisions, such as large register file, data-parallel select, and large basic blocks.

A simpler microarchitecture (left center) reduces area use, design complexity, and critical decision paths, which leads to increased computational density, a high operating frequency, and a short pipeline. The simpler microarchitecture improves the efficiency of static scheduling by reducing the constraints on instruction scheduling and shortening pipeline latencies that the schedule must cover. Instruction bundling simplifies the microarchitecture by streamlining instruction delivery. A simpler microarchitecture in turn eliminates complex rules on instruction placement and thus makes bundling more efficient. Instruction bundling benefits from static scheduling to schedule instructions properly and in turn provides an efficient encoding of statically scheduled instructions.

Wide data paths simplify the microarchitecture by efficiently exploiting data-level parallelism expressed using SIMD instructions. The use of SIMD instructions reduces the total number of instructions and avoids the need to build wider issue architectures to map data-level parallelism onto instruction level parallelism. Finally, instruction bundles map efficiently onto the wide instruction delivery data path.

The local store aids static scheduling by providing low latency, deterministic memory access. It also simplifies the microarchitecture by eliminating tag-match compare and late hit-miss detection, miss recovery, and coherence management associated with cache architectures.

Many design decisions in Figure 8 provide additional synergistic reinforcements. For example, static scheduling magnifies the benefits of the large register file, and the large register file makes it possible to generate better schedules by giving the compiler (or programmer) more instruction scheduling freedom and, indirectly, by providing more registers for advanced optimizations that target instruction-level parallelism (ILP optimizations).

The large register file also exploits data-parallel select operations by providing registers for if-conversion. Exploiting data-parallel select helps code efficiency by supporting data-parallel execution of conditional program flow; building larger basic blocks, which benefit other ILP optimizations; and exploiting sequential fetch. Sequential fetch with large basic blocks, in turn, is key for the effectiveness of sequential fetch with the wide local store port, which allows efficient sharing of the single port. Sharing a single port then contributes to the local store's efficiency.

Synergistic processing clearly drives Cell's performance. The streamlined architecture provides an efficient multithreaded execution environment for both scalar and SIMD threads and represents a reaffirmation of the RISC principles of combining leading edge architecture and compiler optimizations. These design decisions have enabled the Cell BE to deliver unprecedented supercomputer-class compute power for consumer applications.　MICRO

....

References

1. J.A. Kahle et al., "Introduction to the Cell Multiprocessor," *IBM J. Research and Development,* vol. 49, no. 4/5, July 2005, pp. 589-604.

2. V. Salapura et al., "Power and Performance Optimization at the System Level," *Proc. ACM Int'l Conf. Computing Frontiers 2005*, ACM Press, 2005, pp. 125-132.

3. V. Srinivasan et al., "Optimizing Pipelines for Power and Performance," *Proc. 35th Int'l Symp. Microarchitecture*, IEEE CS Press, 2002, pp. 333-344.

4. P. Hofstee, "Power Efficient Processor Architecture and the Cell Processor," *Proc. 11th Int'l Symp. High-Performance Computer Architecture,* IEEE CS Press, 2005, pp. 258-262.

5. K. Diefendorff et al., "Altivec Extension to PowerPC Accelerates Media Processing,"

IEEE Micro, vol. 20, no. 2, Mar. 2000, pp. 85–95.

6. D. Pham et al., "The Design and Implementation of a First Generation Cell Processor," *Proc. Int'l Solid-State Circuits Conf. Tech. Digest*, IEEE Press, 2005, pp. 184-185.

7. A. Eichenberger et al., "Optimizing Compiler for the Cell Processor," *Proc. 14th Int'l Conf. Parallel Architectures and Compilation Techniques*, IEEE CS Press, 2005, pp. 161-172.

8. S. Larsen and S. Amarasinghe, "Exploiting Superword Parallelism with Multimedia Instructions Sets," *Proc. 2000 ACM SIGPLAN Conf. Programming Language Design and Implementation*, ACM Press, 2000, pp. 145-156.

9. S.A. Mahlke et al., "Characterizing the Impact of Predicated Execution on Branch Prediction," *Proc. 27th Int'l Symp. Microarchitecture*, ACM Press, 1994, pp. 217–227.

10. M. Gschwind, "Chip Multiprocessing and the Cell Broadband Engine," to appear in *Proc. ACM Int'l Conf. Computing Frontiers 2006*.

Michael Gschwind is a microarchitect and design lead for a future system at IBM T.J. Watson Research Center, where he helped develop the concept for the Cell Broadband Engine Architecture, was a lead architect in defining the Synergistic Processor architecture, and developed the first Cell BE compiler. His research interests include power-efficient high-performance computer architecture and compilation techniques. Gschwind has a PhD in computer science from Technische Universität Wien. He is an IBM Master Inventor and a senior member of the IEEE.

H. Peter Hofstee is the chief scientist for the Cell BE and chief architect of Cell's synergistic processor element architecture at IBM Austin, and helped develop the concept for the Cell Broadband Engine Architecture. His research interests are future processor and system architectures and the broader use of the Cell. Hofstee has a Doctorandus in theoretical physics from the University of Groningen and a PhD in computer science from the California Institute of Technology.

Brian Flachs is architect, microarchitect, and SPU team lead at IBM Austin. His research interests include in low-latency, high-frequency processors, computer architecture in general, image processing, and machine learning. Flachs has a PhD from Stanford University.

Martin Hopkins is retired after 30 years at IBM. He was a pioneer in the development of optimizing compilers using register coloring and multiple target architectures, which led to work on the instruction set for the IBM 801, the first RISC computer. He is an IBM Fellow.

Yukio Watanabe is with Toshiba Corp., where he designs and develops high-performance microprocessors. Watanabe has a BE in electronic engineering and an ME in information science, both from Tohoku University, Sendai. While assigned to the STI design center in Austin, he helped develop the Cell processor.

Takeshi Yamazaki is a member of the Microprocessor Development Division at Sony Computer Entertainment. His work includes processor and system architecture definition with a focus on scalability and controlability. Yamazaki has a BS and an MS in computer science and a PhD in electronics and information engineering from University of Tsukuba. He is a member of the IEEE.

Direct questions and comments about this article to Michael Gschwind, IBM T.J. Watson Research Center, PO Box 218, Yorktown Heights, NY 10598; mkg@us.ibm.com.

For further information on this or any other computing topic, visit our Digital Library at http://www.computer.org/publications/dlib.

Coming in

May–June 2006

High-Performance

On-Chip

Interconnects