# Using advanced compiler technology to exploit the performance of the Cell Broadband Engine™ architecture

A. E. Eichenberger
J. K. O'Brien
K. M. O'Brien
P. Wu
T. Chen
P. H. Oden
D. A. Prener
J. C. Shepherd
B. So
Z. Sura
A. Wang
T. Zhang
P. Zhao
M. K. Gschwind
R. Archambault
Y. Gao
R. Koo

The continuing importance of game applications and other numerically intensive workloads has generated an upsurge in novel computer architectures tailored for such functionality. Game applications feature highly parallel code for functions such as game physics, which have high computation and memory requirements, and scalar code for functions such as game artificial intelligence, for which fast response times and a full-featured programming environment are critical. The Cell Broadband Engine™ architecture targets such applications, providing both flexibility and high performance by utilizing a 64-bit multithreaded PowerPC® processor element (PPE) with two levels of globally coherent cache and eight synergistic processor elements (SPEs), each consisting of a processor designed for streaming workloads, a local memory, and a globally coherent DMA (direct memory access) engine. Growth in processor complexity is driving a parallel need for sophisticated compiler technology. In this paper, we present a variety of compiler techniques designed to exploit the performance potential of the SPEs and to enable the multilevel heterogeneous parallelism found in the Cell Broadband Engine architecture. Our goal in developing this compiler has been to enhance programmability while continuing to provide high performance. We review the Cell Broadband Engine architecture and present the results of our compiler techniques, including SPE optimization, automatic code generation, single source parallelization, and partitioning.

## INTRODUCTION

The Cell Broadband Engine** (BE) processor provides both flexibility and high performance. The first generation Cell BE processor includes a 64-bit multithreaded PowerPC* processor element (PPE) with

two levels of globally coherent cache. For additional performance, the Cell BE processor includes eight synergistic processor elements (SPEs), each containing a synergistic processing unit (SPU). Each SPE consists of a processor designed for streaming workloads, a local memory, and a globally coherent DMA engine. Computations are performed by 128-bit-wide single instruction multiple data (SIMD) functional units. An integrated high-bandwidth bus connects the nine processors and their ports to external memory and I/O.

The intricacy of the Cell BE processor spans multiple dimensions, each presenting its own set of challenges for both the highly skilled application developer and a highly optimizing compiler. At the elementary level, the Cell BE system has two distinct processor types, each with its own application-level instruction-set architecture (ISA). One ISA (for the PPE) is the familiar 64-bit PowerPC with a vector multimedia extension unit (VMX); the other (for the SPEs) is a new 128-bit SIMD instruction set for multimedia and general floating-point processing. The first Cell BE releases consist of one PPE and 8 SPEs, each with its own 256-KB local memory to accommodate both program instructions and data. Typical applications on the Cell BE processor consist of a variety of code to exploit both of these processors.

The most basic level of programming support for the Cell BE platforms consists of two separate compilers, one targeting the PPE and the other targeting the SPEs, along with a set of utilities and runtime support for loading and running code on the SPEs and transferring data between the system memory and the local stores of the SPEs. It has been demonstrated that very competitive performance can be achieved with the deployment of a low-level programming model, but to make the architecture interesting and accessible to a more general user community, it is useful to abstract the details and present a higher-level view of the system. This issue is addressed by providing a highly optimized compiler for the Cell BE architecture.

IBM has long provided state-of-the-art compiler support for the PowerPC platform, including automatic and user-directed exploitation of shared-memory parallelism. We use this same compiler technology to exploit the performance potential of the Cell BE architecture. The prototype compiler that we have developed for the Cell BE platform generates code, within a single compilation and under option control, for either the PPE or the SPEs, or both. The PPE path of the prototype is essentially the existing PowerPC compiler, complete with VMX support and tuned for the PPE pipeline. For the SPEs, a new path has been developed to target the specific architectural features of this attached processor, including automatic exploitation of the four-way SIMD units. The prototype compiler innovatively takes advantage of and extends existing parallelization technology to enable partitioning and parallelization across multiple heterogeneous processing elements from within a single compilation process. We also draw on the large body of existing research on programming restructuring techniques to automate and optimize data transfer between the multiple processing elements of the system. Our work extends previous research in taking into account not only the heterogeneity of the multiple processing elements but also the nature of the small attached local memories, which are designed to handle both code and data.

When compiling for the most elementary level of the Cell BE architecture, the pipelines of both processors must be taken into account. The SPEs present several challenges not seen in the PPE, chief among them instruction prefetch capabilities and the significant branch miss penalties resulting from the lack of hardware branch prediction. To achieve high rates of computation at moderate costs in power and area, functions that are traditionally handled in hardware, such as memory realignment, branch prediction, and instruction fetches, have been partially offloaded to the compiler. Our techniques address these new demands on the compiler. In the section "Optimized SPE code generation," we discuss in detail the following optimizations: generating scalar code on SIMD units, optimizing language-dictated conversions (i.e., those required by a particular programming language) to increase computations on subwords (i.e., data that is smaller than a word), reducing the performance impact of branches through branch hinting and branch elimination, and scheduling instructions in the presence of limited hardware support for dual issuing and instruction fetching.

At the next level of complexity, the SPE is a short SIMD or multimedia processor, which was not designed for high performance with scalar code.

Although the compiler does support explicit programming of the SIMD engine by means of intrinsics (i.e., functions that are built into the compiler as opposed to those contained in libraries), it also provides the novel *auto-SIMDization* functionality, which generates vector instructions from scalar source code for the SPEs and the VMX units of the PPE. Auto-SIMDization is the process of extracting SIMD parallelism from scalar loops. In the section "Generation of SIMD code," we describe auto-SIMDization in some detail, including how it minimizes overhead due to misaligned data streams and how it is tailored to handle many of the code structures found in multimedia and gaming applications.

Using the parallelism of the Cell BE processor when deploying applications across all its processing elements, our compiler enhances its programmability by parallelizing and partitioning a single source program across the PPE and the eight SPEs, guided by user directives. The compiler also efficiently uses the complex memory system that ties all these processors together on the chip and interfaces with the external storage. While the PPE makes use of a conventional two-level cache, each SPE draws data and instructions from its own small memory, internal to the chip. Data transfers to and from the local stores must be explicitly managed by using a DMA engine. Within the compiler, we have developed techniques to generate and optimize the code that accomplishes data transfer, allowing a single SPE to process data that far exceeds the local store's capacity, using code that also exceeds the size of its local store, and scheduling the necessary transfers so that they overlap ongoing computation to the extent that this is achievable. In the section "Optimized SPE code generation," we discuss the compiler's generation of parallel code and describe our code-partitioning techniques.

Our goal in developing this compiler has been to enhance the programmability of the architecture, at the same time continuing to provide respectable performance. Currently average speedup factors of 1.3, 9.9, and 6.8 for our SPE, SIMD, and parallelization compilation techniques are demonstrated on suitable benchmarks, indicating some initial success with our approach. In the section "Measurements," we briefly review our current performance measurements, and we conclude in the following section.

## CELL BE ARCHITECTURE

The implementation of the first-generation Cell BE processor[1] includes a Power Architecture processor and eight attached processor elements connected by an internal, high-bandwidth Element Interconnect Bus (EIB). ***Figure 1*** shows the organization of the Cell BE elements.

The PPE consists of a 64-bit, multithreaded Power Architecture processor with two levels of on-chip cache. The cache preserves global coherence across the system. The processor also supports IBM's VMX[2] to accelerate multimedia applications by using VMX SIMD units.

A major source of computing power is provided by the eight on-chip SPEs.[3] An SPE consists of a new processor designed to accelerate media and streaming workloads, its local noncoherent memory, and its globally coherent DMA engine. The units of an SPE and key bandwidths are shown in Figure 1.

Most instructions operate in a SIMD fashion on 128 bits of data representing either two 64-bit double-precision floating-point numbers or longer integers, four 32-bit single-precision floating-point numbers or integers, eight 16-bit subwords, or sixteen 8-bit characters. The 128-bit operands are stored in a 128-entry unified register file. Instructions may take up to three operands and produce one result. The register file has a total of six read and two write ports.

The memory instructions also access 128 bits of data, with the additional constraint that the accessed data must reside at addresses that are multiples of 16 bytes. Thus, when addressing memory with vector load or store instructions, the lower four bits of the byte addresses are simply ignored. To facilitate the loading and storing of individual values, such as a character or an integer, there is additional support to extract or merge an individual value from or into a 128-bit register.

An SPE can dispatch up to two instructions per cycle to seven execution units that are organized into even and odd instruction pipes. Instructions are issued in order and routed to their corresponding even or odd pipe by the issue logic, that is, a component which examines the instructions and determines how they are to be executed, based on a number of constraints. Independent instructions are detected by the issue logic and are dual-issued (i.e.,

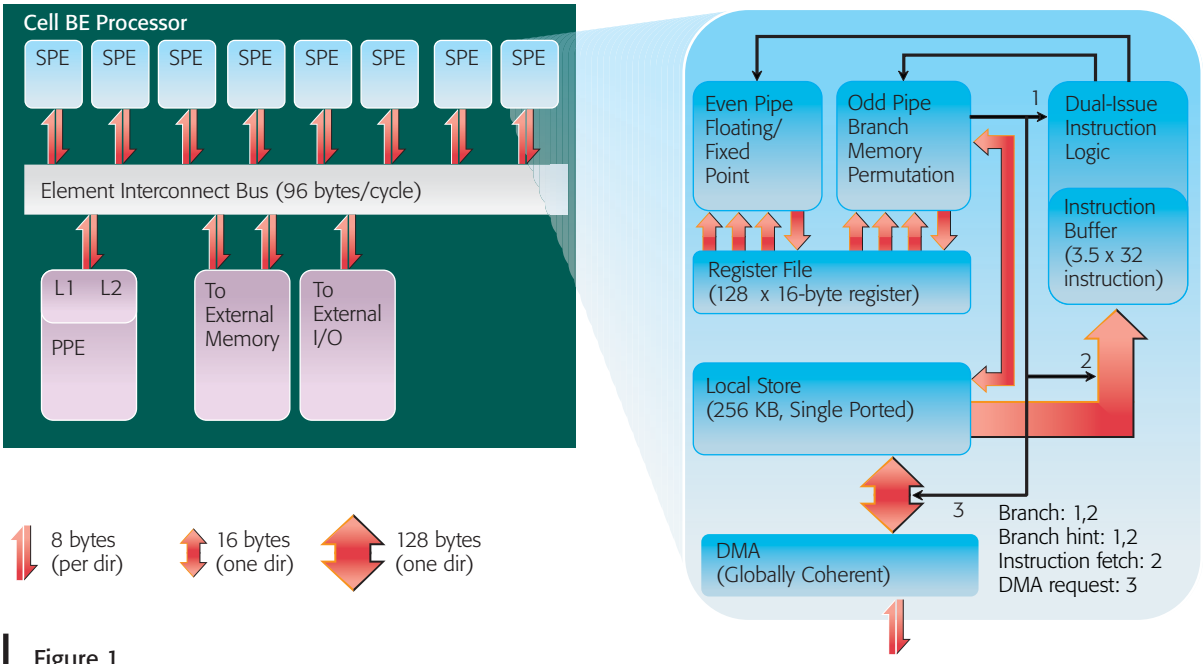


**Figure 1**
Organization of Cell BE elements

dispatched two per cycle) provided they satisfy the following condition: the first instruction must come from an even word address and use the even pipe, and the second instruction must come from an odd word address and use the odd pipe. When this condition is not satisfied, the two instructions are executed sequentially. The instruction latencies and their pipe assignments are shown in *Table 1*.

The SPE's 256-KB local memory supports fully pipelined 16-byte accesses (for memory instructions) and 128-byte accesses (for instruction fetches and DMA transfers). Because the memory has a single port, instruction fetches, DMA, and memory instructions compete for the same port. Instruction fetches occur during idle memory cycles, and up to 3.5 fetches may be buffered in the instruction fetch buffer to better tolerate bursty peak memory usage. The maximum capacity of the buffer is thus 112 32-bit instructions. An explicit instruction can be used to initiate an inline instruction fetch.

**Table 1.** Latencies and pipe assignment for SPE

| Instruction | Pipe | Latency (cycles) |
|---|---|---|
| arithmetic, logical, compare, select | even | 2 |
| byte sum/diff/average | even | 4 |
| shift/rotate | even | 4 |
| float | even | 6 |
| integer multiply-accumulate | even | 7 |
| shift/rotate, shuffle, estimate | odd | 4 |
| load, store | odd | 6 |
| channel | odd | 6 |
| branch | odd | 1–18 |

The SPE hardware assumes that branches are not taken, but the architecture allows for a "branch hint" instruction to override the default branch prediction policy. In addition, the branch hint instruction causes a prefetch of up to 32 instructions, starting from the branch target, so that a branch taken according to the correct branch hint incurs no penalty. One of the instruction fetch buffers is reserved for the branch-hint mechanism. In addition, there is extended support for eliminating short branches by using select instructions.

Data is transferred between the local memory and the DMA engine in units of 128 bytes. The DMA engine can support up to 16 concurrent requests of up to 16 KB originating either locally or remotely. The DMA engine is part of the globally coherent memory address space; addresses of local DMA requests are translated by an MMU (memory management unit) before being sent on the bus. Bandwidth between the DMA and the EIB bus is 8

bytes per cycle in each direction. Programs interface with the DMA unit through a channel interface and may initiate blocking as well as nonblocking requests.

## OPTIMIZED SPE CODE GENERATION

In this section, we describe the current compiler optimization techniques that address key architectural features of the SPE. A user interested in SPE code generation may observe that our SPE compiler produces the high quality code normally associated with the XL compiler suite.

### Scalar code on SIMD units

As mentioned in the section "Cell BE architecture," most SPE instructions are SIMD instructions operating on 128 bits of data at a time, including all memory instructions. One notable exception is the conditional branch instruction, which branches on nonzero values from the primary slot (i.e., the highest order or leftmost 32 bits) of a 128-bit register. The address fields are also expected by memory instructions to reside in primary slots.

When scalar code is generated on an SPE, it is critical that the SIMD nature of the processor does not get in the way of program correctness. For example, an `a = b + c` integer computation on a scalar processor simply requires two scalar loads, one add, and one store instruction. When executing on the SPE, a load of `b` yields a 128-bit register value, which contains the 32-bit value of `b` (this is technically true only when the data elements are naturally aligned, as we assume them to be in this paper). Its actual location within the 128-bit register is determined by the 16-byte alignment of `b` in local memory. This is true because the memory subsystem performs only 16-byte-aligned memory requests.

After values are loaded in registers, the compiler must track the alignment of the data because it may operate only on values that have the same relative alignment. In our `a = b + c` example, the 128-bit registers may be added only if the location of the `b` and `c` values in their respective registers is identical. When this is not the case, the compiler must permute the contents of one of the registers to match the alignment of the other. Because scalar computations in highly optimized multimedia codes mostly involve address and branch-condition computations (which must reside in the primary slot when used by the memory and branch instructions), the default policy is to move any misaligned scalar data into the primary slot.

The storing of a scalar value is also not as straightforward as on a scalar processor. The compiler must first load the original 128-bit data in which the result resides, variable `a` in our example, then splice the new value in the original data, and finally store the resulting 128-bit data to the local memory.

Without special care, a worst case scenario for our `a = b + c` example could result in two load and permute instructions to get and align the input data, one add instruction to compute the result, and one load, permute, and store instruction to store the result in memory. We take several steps to avoid such overhead. First, we allocate all local and global scalars to their own private 128-bit local memory lines and align the scalars into their primary slots. Although this results in some memory overhead, it is insignificant compared to the increase in code size generated by the extra permutation instructions that would otherwise be needed to realign the data. Second, we perform aggressive register allocation of all local computations, such as address and loop index variables, to make good use of the 128-entry register file. As a result, such variables often reside exclusively in the primary slot of registers and thus need no memory storage and associated load and store instructions. Finally, auto-SIMDization is applied to the code so as to minimize the remaining scalar code in an application.

### Subword optimization

The SPE instruction set natively supports operations on a wide range of data widths, from 8-bit bytes to 64-bit doublewords, unlike most RISC (Reduced Instruction Set Computer) processors, which typically support operations on words or doublewords only. Because programming languages were designed with traditional processors in mind, languages typically promote all the computations of the short data type (e.g., 8-bit characters ["chars"] and 16-bit short variables ["shorts"], referred to as "subwords" here) to integers. As discussed next, such promotions have a negative performance impact on the SPE. Subword optimization attempts to alleviate this performance impact.

To illustrate this, we examine an example of `a = b · c`, where all variables are declared as 16-bit shorts. The integral promotion rule in the C programming language requires any subword types be automatically promoted to the integer type before performing

any computations thereon. Effectively, C allows programmers to specify subword data types in terms of alignment, size, and layout in memory, but does not allow specifying subword arithmetic. In essence, shorts and chars in C are simply a compressed representation of integers in memory, which are decompressed when loaded from memory.

Such compression and decompression have little performance impact on traditional RISC processors, because their scalar memory operations typically perform such size and sign extensions "on the fly." This is not the case with SIMD units such as those in the SPE because they are specifically designed to operate on subword data types in a compressed format within their registers. This is precisely the feature that allows them, for example, to simultaneously operate on eight shorts packed in a 128-bit register. Because sign extensions are not part of the load instructions, they have to be handled explicitly by additional instructions on the SPE. When applied to the `a = b · c` example, the integral promotion rule thus requires two additional sign extension instructions to promote the `b` and `c` input variables.

Another cost factor is that the SPE, like many other SIMD units, does not support each data type equally. For example, it provides hardware support only for 16-bit integer multiply-add instructions. Hence, a 32-bit multiply instruction is supported in software by using a series of 16-bit multiply-add instructions, three in our example. This is an inherently wasteful computation because ultimately only the lower 16 bits of the result are stored in memory.

Subword arithmetic optimization bridges the gap between C's inability to specify subword arithmetic operations and the SPEs' underlying strength in supporting subword operations. In our work, we implemented subword optimization in two steps. The first step eliminates the redundant partial-copy operations. Instead of solely focusing on redundant sign extensions, we propose a more general framework based on the concept of partial-copy propagation. We exploit the fact that some instructions (such as subword instructions) use only a subset of their register values, whereas other instructions (such as sign extension instructions) copy only a subset of their register values. This framework eliminates redundant sign extension operations, performs folding (the process of collapsing multiple instructions into one), and is easily extendable.

The second step generates the subword arithmetic operations. This optimization aims at converting integer arithmetic operations to equivalent arithmetic operations of narrower data width. This optimization can produce more efficient code when a word instruction can be replaced with a more efficient subword instruction. When subword and word instructions are equally supported, narrowing down the operating data width reveals more opportunity for folding, for example by eliminating redundant sign extensions.

### Branch optimizations

The SPEs are heavily pipelined, making the penalty for incorrect branch prediction high, namely 18 cycles. In addition, the hardware's branch prediction policy is simply to assume that all branches (including unconditional branches) are not taken. In other words, branches are only detected late in the pipeline at a time where there are already multiple fall-through instructions (i.e., those in the sequential path) in progress. This design achieves reduced hardware complexity, faster clock cycles, and increased predictability, which is important for multimedia applications.

Because branches that are taken are so much more expensive than the fall-through path, the compiler first attempts to eliminate taken branches. One effective approach for short "if-then-else" constructs is "if-conversions," which use compare-and-select instructions provided by the SPE to avoid branching code. Another approach is to determine the likely outcome of branches in a program, either by means of compiler analysis or through user directives, and perform code reorganization techniques to move "cold" paths (i.e. those unlikely to be taken) out of the fall-through path.

However, many taken branches cannot practically be eliminated, in cases such as function calls, function returns, loop-closing branches, and some unconditional branches. To boost the performance of such predictably taken branches, the SPE provides for a branch hint instruction, referred to as "hint for branch" or `hbr`. This instruction specifies the location of a branch and its likely target address. When the `hbr` instruction is scheduled sufficiently early (at least 11 cycles before the branch), instructions from the hinted branch target are prefetched from memory and inserted in the instruction stream immediately after the hinted

branch. When the hint is correct, the branch latency is essentially one cycle; otherwise, the normal branch penalty applies. Presently, the SPE supports only one active hint at a time.

Likely branch outcomes can either be measured through branch profiling, estimated statistically by means of sets of heuristics, or be provided by the user (we currently use the latter technique). We then insert a branch hint for branches with a probability of being taken which is higher than a given threshold.

For loop-closing branches, we attempt to move the `hbr`s outside the loop to avoid the repetitive execution of the hint instruction. This optimization is possible because a hint remains in effect until replaced by another one. Because an `hbr` instruction indicates the address of its hinted branch by a relative, 8-bit signed immediate field, an `hbr` and its branch instruction must be within 256 instructions of each other. Thus, `hbr` instructions can only be removed from small- to medium-sized loops. Furthermore, we can move the hint outside of a loop only when there are no hinted branches inside the loop body because at most one hint can be outstanding at a time.

Unconditional branches are also excellent candidates for branch hint instructions. The indirect form of the `hbr` instruction is used before function returns, function calls using pointers, and all other situations that give rise to indirect branches.

## Instruction scheduling

The scheduling process consists of two closely interacting subtasks: scheduling and bundling. The *scheduling* subtask reorders instructions to reduce the length of critical paths. The *bundling* subtask ensures that the issue constraints are satisfied to enable dual issuing and prevent instruction fetch starvation, i.e., the situation when the instruction buffer is empty and awaiting refill from an explicit instruction fetch (said to be ifetch but, technically speaking, `hbr.p`) instruction. We first describe these two subtasks and investigate their interaction in more detail.

The scheduler's main objective is to schedule operations that are on a critical path with the highest priority and schedule the other less critical operations in the remaining slack (i.e., idle) time. It ensures that instructions that are expected to be dual-issued have no dependence and resource conflict. Typical schedulers deal only with constraints concerning resources and latencies. On the SPEs, however, there are some constraints involving numbers of instructions; for example, the constraint that an `hbr` branch hint instruction cannot be more than 256 instructions nor less than eight instructions from its target branch. Constraints expressed in terms of instruction counts are further complicated by the fact that the precise number of instructions in a scheduling unit is known only after the second, bundling subtask has been completed.

The bundler's main role is to ensure that each pair of instructions that is expected to be dual-issued satisfies the SPEs' instruction issue constraints. As mentioned in the section "Cell BE architecture," the hardware dual-issue constraint states that the first instruction must use the even pipe and reside at an even word address, whereas the second instruction must use the odd pipe and reside at an odd word address. After the instruction ordering is set by the scheduling subtask, the bundler can impact the evenness of the word address of a given instruction only by judiciously inserting nop (null operation) instructions into the instruction stream.

Another important task of the bundler is to prevent instruction fetch starvation. Because the single local memory port is shared between the instruction fetch mechanism and the processor's memory instructions, a large number of consecutive memory instructions can stall instruction fetching. With 2.5 instruction fetch buffers reserved for the fall-through path, the SPEs can starve for instructions in 40 dual-issued cycles. After a fetch buffer is empty (this takes 16 dual-issued cycles or more), there may be a window as small as nine dual-issued cycles in length in which the empty buffer can be refilled in order to hide the full 15-cycle instruction fetch latency. Thus, the bundling process must keep precise information at compile time about the status of the instruction fetch buffers, mainly by keeping a precise count of the numbers of instructions already bundled in the function (for simplicity, the first instruction of a function is laid out in memory to reside at an instruction fetch boundary, that is, a multiple of 16-word addresses). Using this instruction count, the compiler can determine when an instruction fetch buffer is becoming empty and whether an explicit instruction fetch will be needed to prevent starva-

tion because of a burst in memory traffic. The ifetch instruction explicitly fills an instruction fetch buffer along the fall-through path.

This buffer-refilling window is even smaller after a correctly hinted branch because then there is only a single valid instruction fetch buffer (the one that was prefetched by the `hbr` instruction), as opposed to 2.5 buffers for the fall-through path. In such cases, the instruction window is so small that we must further ensure that all instructions in the prefetched instruction buffer are part of the execution path associated with the taken branch. In other words, the branch target must point to code starting at an address that is a multiple of 16 instructions, which is the unit of realignment of the instruction fetch mechanism. To enforce this alignment constraint, we may need to introduce `nop` instructions. Our heuristics are fairly successful at utilizing any idle time slots so that `nop` instructions may be inserted without a performance penalty.

A final concern of the bundling process is to make sure that there are a sufficient number of instructions between a branch hint and its branch instruction. This constraint is due to the fact that a hint is only fully successful if its target branch address is computed before that branch enters the instruction decode pipeline. The bundler adds extra `nop` instructions when the scheduler does not succeed in interleaving a sufficient number of independent instructions between a hint and its branch.

Our initial approach was to keep the scheduling and bundling separate and perform them in that order; however, this had a negative performance impact for tight loops with critical computational requirements and bursty memory behavior. In such cases, the bundler frequently added an ifetch instruction to break a critically long series of memory accesses, thus inserting an extra cycle with an instruction that uses the memory unit on the odd pipe. Because bundling was performed after scheduling, the idle even pipe unit (on which a critical computation could have be scheduled) could not be put to good use, as the schedule was already fixed.

For this and similar occurrences, our current scheme uses a unified scheduling and bundling phase. When preparing to schedule the next, empty cycle in the scheduling unit, we first investigate if an ifetch instruction is required. When this is the case, we force an ifetch instruction in that cycle and update the scheduling resource model accordingly. We then proceed with the normal scheduling process for that cycle. When no additional instruction can be placed in the current cycle, we investigate if nop instructions must be inserted in prior cycles to enable dual issuing. After this task is completed, we proceed to the next cycle. We generally preserve the cyclic scheduling approach, except that we may retroactively insert nop or ifetch instructions, as required by the bundling process.

## PROGRAMMING FOR THE CELL BE ARCHITECTURE

In this section, we focus on how the compiler may be used as one of the tools in developing applications for the Cell BE architecture. In the existing programming model for the Cell BE architecture, the heterogenous processor cores, the SPEs and the PPE, and the non-uniform memory accesses of the architecture are visible to application programmers. Efficient manual programming of the Cell BE architecture can be a complex task. To make use of the best features of the PPE and SPE cores for an application, programmers must manually partition the application into separate code segments and use the compiler that targets the appropriate ISA.

As previously described, both the VMX unit of the PPE and the SPE are SIMD processors. Efficient programming therefore requires exploiting parallelism within the SPE and the VMX unit by using the rich set of vector intrinsics that are provided for each. To further exploit the coarse-grained interprocessor parallelism throughout the Cell BE system, the programmer may choose to partition his application into tasks or parallel work units that may be executed on the SPEs by using a pipeline or parallel-execution model.

The resulting PPE and SPE code segments must work together cooperatively and must explicitly manage the transfer of code and data between system memory and the limited SPE local stores. Optimizing data transfer to overlap communication and computation may involve manually programming multibuffering schemes that take into account the optimal size and numbers of local data buffers and that select the best placement of data transfer requests. The extent to which parallelism is deployed in the application also influences the data transfer decisions.

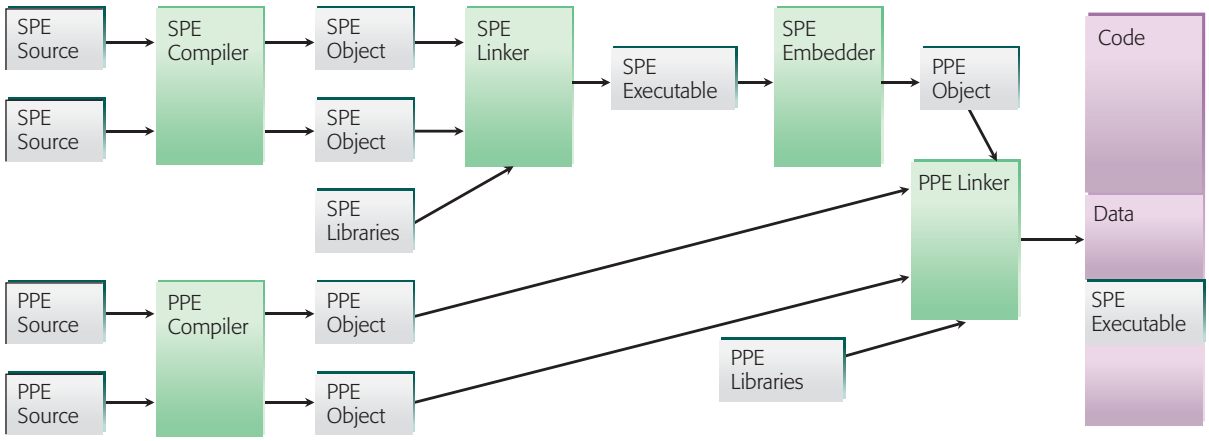


**Figure 2**
Manually compiling and binding a Cell BE program

The final step to effect the seamless execution of a Cell BE program requires using the SPE linker and an embedding tool to generate a PPE executable that contains the SPE binary embedded within the data section. This object is then linked, using a PPE linker, with the runtime libraries which are required for thread creation and management, to create a bound executable for the Cell BE program. ***Figure 2*** demonstrates a manual process for creating such an executable.

Although manually partitioning code and data into PPE and SPE portions and explicitly managing the transfers of code and data between system memory and local stores may be a common approach to programming the Cell BE architecture (and indeed may well be the preferred approach of the expert programmer for extracting the maximum performance), we believe that in many instances this imposes too great a burden on the programmer. An important focus of our work is the deployment of sophisticated compiler technology to simplify programming for the Cell BE architecture, thereby enabling its use in a more general-purpose environment, while still providing the expert programmer with full control and access to the high performance code generated by our compiler. To accomplish this, we have provided an incremental approach, which delivers increasing levels of compiler support for a broad range of programming styles and expertise levels.

An expert programmer may choose to program directly in assembly language. This approach may yield the highest performance when used appropriately, but incurs a significant cost in terms of application development. To extract high performance from their application with significantly higher ease of use and development productivity, programmers can use our PPE and SPE C compilers with VMX and SPE intrinsics to precisely control SIMD instruction selection and how data is laid out in memory, while letting the compiler schedule the chosen SIMD instructions and allocate them to registers. In the intrinsic support provided for the Cell BE architecture, new data types are introduced to express vectors, using (essentially) a vector type for each of the types supported by the SIMD units, such as vector `char`, `short`, `int`, `long long`, `float`, and `double`. The programmer can then select the appropriate intrinsic with which to operate on the data. Intrinsics appear to the programming language as functions that mimic the behavior of each SIMD instruction in the target architecture. During compilation, these functions are replaced by the actual native SIMD instruction.

The major advantage of programming with intrinsics is that the programmer has full control over the handling of the data alignment and the choice of SIMD instructions, yet continues to have the benefit of high-level transformations such as loop unrolling and function inlining (i.e, copying subroutine code into the calling routine) as well as low-level optimizations such as scheduling, register allocation, and other optimizations discussed in the section "Optimized SPE code generation." In addition, the programmer can rely on the compiler to generate all the scalar code, such as address,

branch, and loop code generation; code that is often error-prone when hand-coded in assembly language. In addition, it is much easier to modify code programmed with intrinsics over the course of an application's development than to modify assembly language. For example, adding a few computations in a program written in assembly language might require retuning the register allocation of an entire function. There are no such issues with intrinsics, as the compiler assigns local temporary variables to registers during the compilation process.

Programmers wishing to achieve good application performance without dealing with the specific SIMD instructions of the target SIMD units (either for portability or application-development cost reasons) may use auto-SIMDization, which extracts SIMD parallelism from scalar loops by first analyzing the data layout in the application and performing optimizations that increase the amount of available SIMD parallelism, and then generating the appropriate SIMD code. Although the compiler can generate SIMD code automatically (regardless of the alignment of the data, for example), the user is encouraged to supply feedback using directives to provide higher-level information to the compiler. The reverse is also true; namely, the compiler can provide high-level feedback on each of the loops on which it has succeeded or failed to perform SIMDIZation. This can be used in turn by the programmer to better tune an application for higher levels of SIMD performance.

For those programmers seeking the highest degree of productivity and ease of use, we provide a level of support which allows a programmer to write an application for the Cell BE architecture without consideration of the intricacies of the heterogeneous ISA and the necessary data transfer. In particular, it is usual for a programmer to view a computer system as possessing a single addressable memory and for all the program data to reside in this space. The compiler provides user-guided parallelization and compiler management of the underlying memories for code and data. When the user directives are applied in a thoughtful manner by a competent user, the compiler provides significant ease of use without significantly compromising performance.

## GENERATION OF SIMD CODE

Prior work in automatic SIMD code generation includes unroll-and-pack approaches[4,5] and loop-based approaches.[6–12] Our current approach combines aspects of both of these approaches. It also attempts to systematically minimize the impact of data reorganization due to compile-time or runtime data misalignment, and it can perform auto-SIMDization in the presence of data conversion (i.e., conversion from one data type to another). Auto-SIMDization can generate such minimum data-reorganization code for both the VMX and SPE SIMD units.

### Example of a loop with misaligned accesses

The following code example of a loop with misaligned accesses illustrates the impact of alignment constraints in this context:

```
for (i=0, i<100; i++) {
  a[i+2] = b[i+1] + c[i+3];
}
```

Assuming for conciseness that each array is 16-byte-aligned, the data involved in a loop iteration, namely `a[i+2]`, `b[i+1]`, `c[i+3]`, are relatively misaligned. The data touched by the first `i=0` loop iteration is highlighted by white circles inside grey boxes in *Figure 3*.The three memory references `b[1]`, `c[3]`, and `a[2]` reside at different locations within their respective 16-byte units of memory. Specifically, they are in the second, fourth, and third integer slots of their respective 4-integer/16-byte unit of memory.

To produce a correct result, this data must be reorganized in registers so that all the data involved in a computation resides in the same integer slot. In Figure 3, we first shift right by one the stream of data generated by `b[i+1]` for `i=0 to 99`. We shift left by one integer slot the stream of data generated by `c[i+3]` for `i=0 to 99`. At this stage, both the `b` and `c` register streams start in the third integer slot. The vector add is then applied to the shifted streams and produces the expected results, `b[1] + c[3], ... , b[100] + c[102]`. To understand the applicability of this scheme, it is critical to realize that "shifting left" and "shifting right" are data reorganizations that operate on a stream of consecutive registers, not the traditional logical or arithmetical shift operation.

### Definitions and valid SIMDizations

The alignment-handling framework used here is based on the concept of streams.[11] A stream represents a sequence of contiguous memory

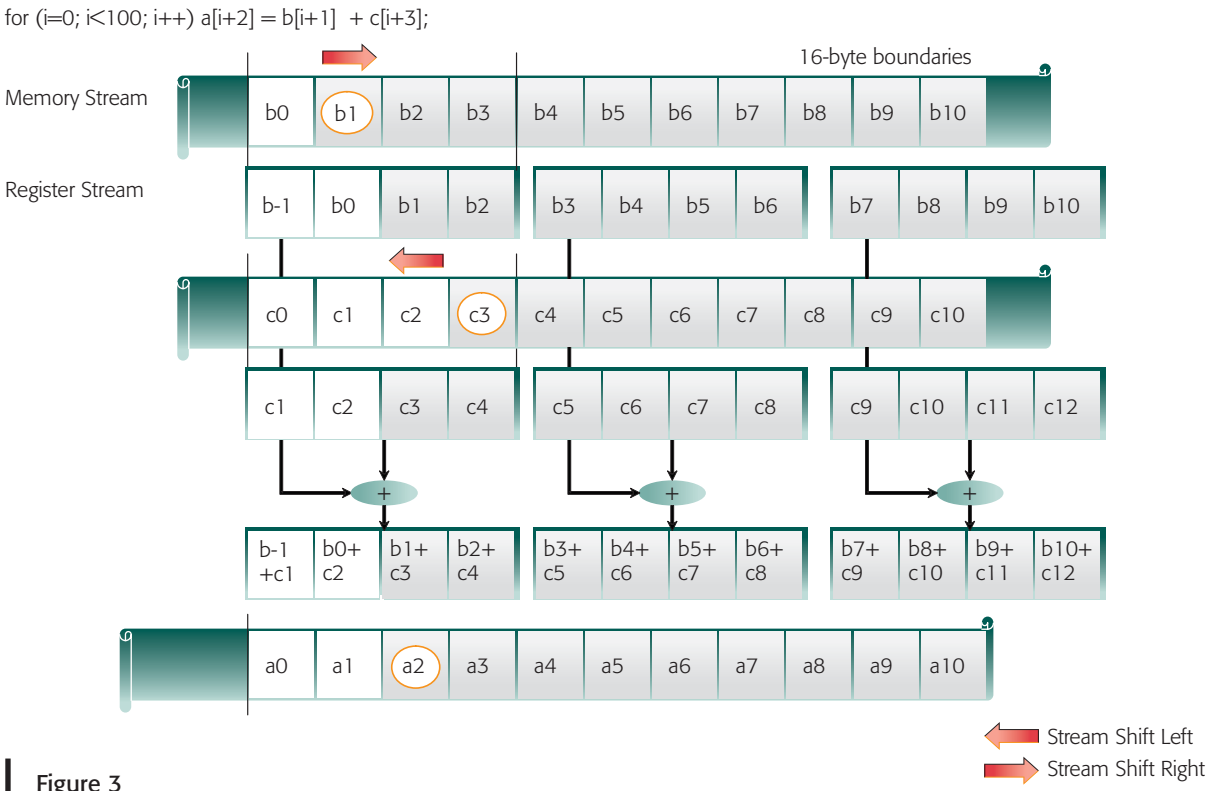


**Figure 3**
Alignment of misaligned accesses

locations that are accessed by a memory reference throughout the lifetime of a loop (shown in Figure 3 as a sequence of grey boxes). By analogy, a stream is also a sequence of contiguous registers that are used by an instruction over the lifetime of a loop.

Instructions in a loop can be viewed as operations on streams. For example, a load operation consumes a stream of memory and produces a stream of registers. An important property of a stream is its *stream offset*, which is defined as the byte offset of the first desired value in the first register of a stream. The offset of a register stream produced by a load operation is the alignment of the first desired value of the input memory stream (namely, the memory address of the first desired value, modulo the vector length of the SIMD unit).

The alignment-handling framework specifies the alignment constraints of a valid SIMDization as follows. In the SIMDization of a store operation, the byte offset of the register stream must match the memory alignment of the memory stream. For a non-unary operation, all the data involved in the computation must reside at the same byte offset in their respective register streams. In the presence of misalignments, a valid SIMDization can only be achieved by judiciously using data reorganization operations to enforce the desired stream offsets. The stream shift operation $\texttt{vshiftstream}(S, c_{in}, c_{out})$ is introduced for this purpose. It shifts all values of a register stream $S$ across consecutive registers of the stream from offset $c_{in}$ to offset $c_{out}$. Figure 3 gives examples of shifting streams left and right.

## Overview of alignment-handling framework

The first phase of the SIMDization framework[13] extracts SIMD parallelism at different program scopes and generates generic vector operations. The next phase transforms the code to satisfy the precise architectural constraints. The final phase converts the generic vector operations to platform-specific instructions.

*Phase 1: Extraction of SIMD parallelism*. This phase first extracts SIMD parallelism within a basic block by placing isomorphic (i.e., identical or similar) computations at adjacent memory locations by using an approach similar to that used in Reference 4. This phase catches manually unrolled loops,

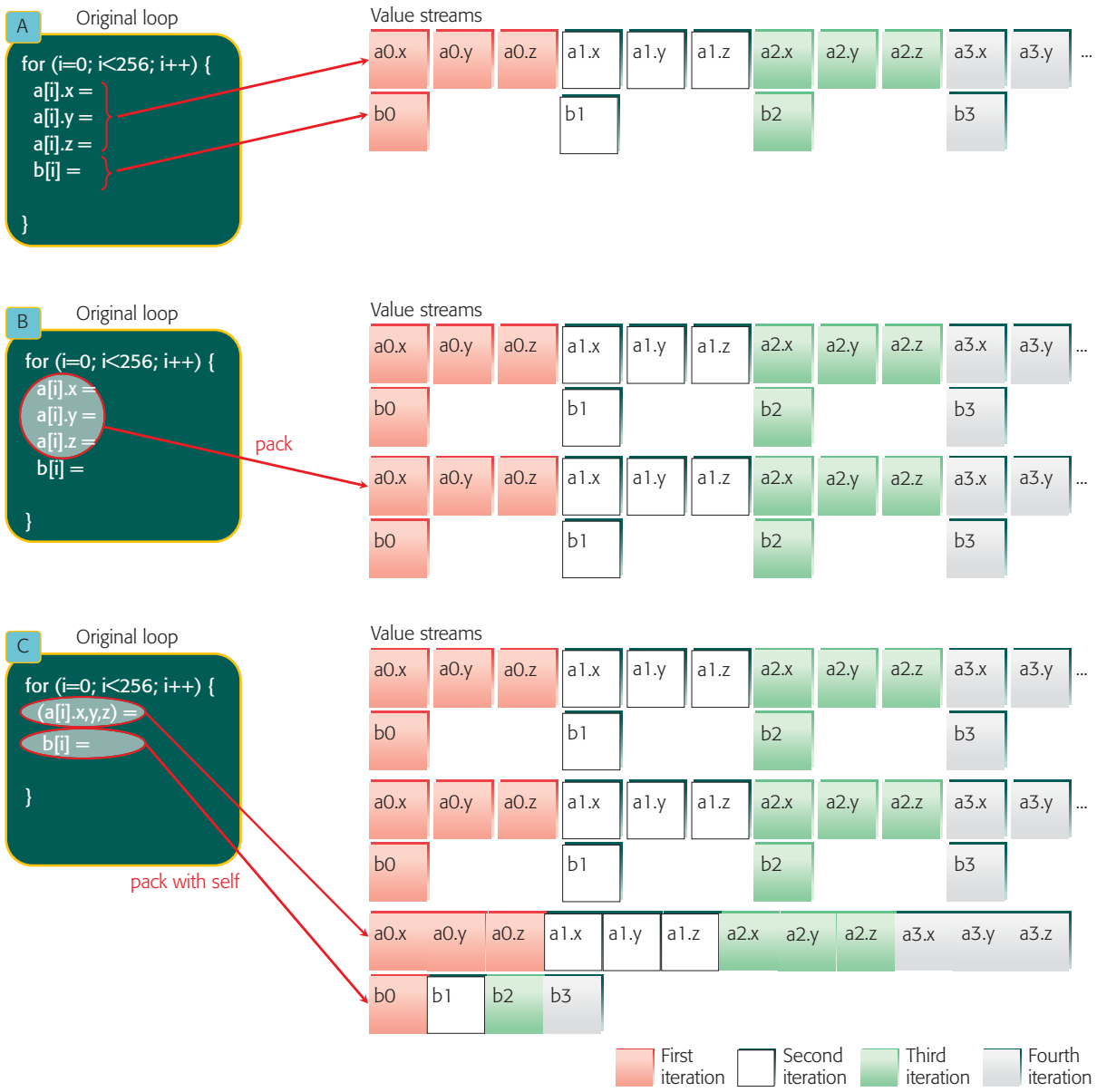


**Figure 4**
Extraction of SIMD parallelism

which are frequently found in multimedia code, and extracts SIMD code among isomorphic or semi-isomorphic computations that involve references such as `a.x`, `a.y`, `a.z`, which are often found in graphic applications, to express three-dimensional coordinates.

SIMD parallelism across loop iterations is then extracted. Iterative computations on stride-one accesses (i.e., those which sequentially access an array element by element) are aggregated into SIMD operations by "blocking the loop." Blocking the loop is a process by which arrays are accessed in sections, rather than by element. The blocking factor (i.e., the size of the sections) is determined such that the byte length of each vector is a multiple of 16 bytes.

The combined extraction of SIMD parallelism within a basic block and among consecutive loop iterations is illustrated in ***Figure 4***. Although the actual algorithm takes all memory references into account,

we focus here only on the store operations generated by each of the loop's four statements. Each individual 4-byte store is represented by a box in Figure 4; the colors of the boxes distinguish consecutive iterations of the original loop.

The compiler recognizes that the first three stores in Figure 4A, `a[i].x`, `a[i].y`, and `a[i].z`, are adjacent in memory. Assuming here that the right-hand sides of the three statements are isomorphic, it aggregates the three statements into a vector of three integers stored as `a[i].x,y,z`, as shown in Figure 4B. The `b[i]` statement remains unchanged. Recognizing the vector store `a[i].x,y,z` and the element store `b[i]` as stride-one accesses, the compiler further aggregates these accesses across loop iterations. In doing so, it treats the new vector `a[i].x,y,z` statement no differently than any other statements in the loop. The only difference between the `a[i].x,y,z` and `b[i]` statements is that the former generates a 12-byte value, whereas the latter generates a 4-byte value.

During this phase, we extract SIMD parallelism among the smallest number of consecutive iterations while ensuring that each vector in the loop has a length that is a multiple of the physical vector length, 16 bytes in this example. We would determine that the optimal blocking factor is 4 here because it aggregates four of the 12-byte `a[i].x,y,z` vectors into a new compound vector of 48 bytes and four of the 4-byte `b[i]` values into a new vector of 16 bytes. The resulting loop is shown in Figure 4C.

*Phase 2: Alignment handling*. This phase takes SIMDized computations as input and inserts shift operations to satisfy the alignment constraints of SIMD operations. The output is a computation augmented with stream-shift operations. During this process, different shift-placement policies can be applied to minimize the number of generated shifts, three of which are described next.

1. *Zero-shift policy*—This policy[8] shifts each misaligned load stream to offset zero and shifts the storage stream from offset zero to the alignment of the store address. (See ***Figure 5A***.) This is the least optimized policy.
2. *Eager-shift policy (ESP)*—This policy[11] shifts each load stream directly to the alignment of the storage stream. (See Figure 5B.) This policy is followed in the example shown in Figure 3.

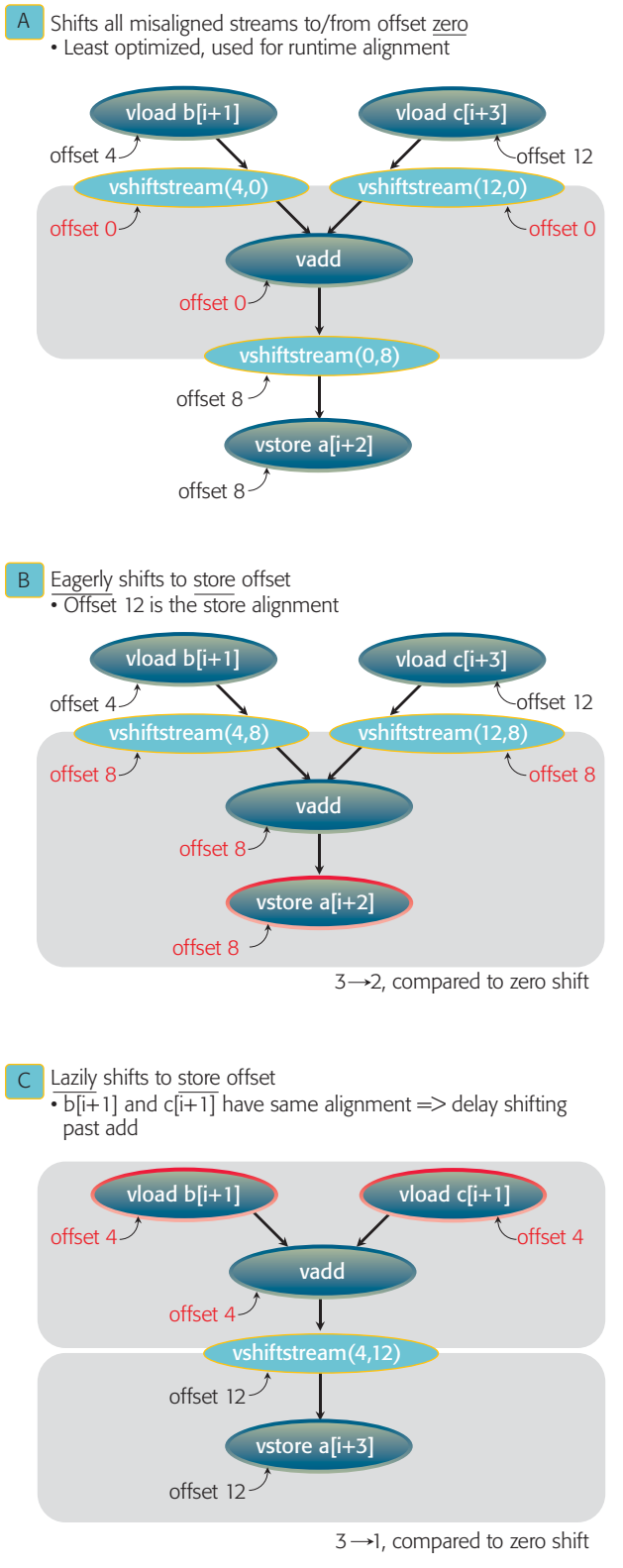


**Figure 5**
Impact of stream-shift policies: (A) zero policy; (B) eager policy; and (C) lazy policy

3. *Lazy-shift policy*—This policy[11] pushes the shift toward the root of the expression tree (i.e., a graphical representation of the expression) as closely as possible. (See Figure 5C.)

In general, ESP inserts fewer stream shifts than the zero-shift policy, for example, two versus three shifts for the loop in the section "Example of a loop with misaligned accesses." ESP initially was only applicable to compile-time alignment[11] but later was extended to runtime alignment as well.[12]

*Phase 3: SIMD code generation*. This phase takes the augmented SIMDized computations as input and maps generic stream-shift operations to native SIMD permutation instructions. For each stream shift in the tree, the algorithm generates a `vperm` instruction in the generated steady-state loop. Specifically, `vperm(v1, v2, L)` selects bytes `L, L+1, . . . , L+V–1` from a double-length vector constructed by concatenating `v1` and `v2`, where `V` is the vector length. For VMX, `vperm` is then mapped to a `vec_perm` VMX instruction.

## GENERATION OF PARALLEL CODE

The SIMDization we have described thus far affords up to peak performance within a single SPE, but the Cell BE architecture enables parallelism in multiple dimensions: the heterogenous PPE and SPE cores, multithreaded PPE, multiple SPEs, and SIMD support in both the PPE and SPEs. Extracting the greatest performance from this architecture necessitates exploiting parallelism across multiple processing elements. We now describe our compiler support for parallel programming across the PPE and multiple SPEs.

Our current approach uses the OpenMP[14] programming model. This provides programmers with the abstraction of a single shared-memory address space. With our prototype compiler, programmers may use OpenMP directives to specify regions of code that can execute in parallel. In this way, they need only write and compile a single body of code, and the compiler takes on the responsibility of duplicating code sections for heterogenous cores (i.e., the PPE and SPE cores) and of coordinating their execution.

### Single source code compilation

We use the existing parallelization infrastructure of the IBM XL compiler for our OpenMP implementation. The XL compiler includes a high-level optimizer called the Toronto Portable Optimizer (TPO). TPO works in two passes: the first pass applies intraprocedural optimizations, and the second pass, also called the "link phase," performs interprocedural optimizations as well.

In the first pass, the compiler outlines each parallel code section; that is, it creates a new function containing a copy of the code in that parallel section and then replaces the original code section with a call to the corresponding function. ***Figure 6*** shows the call graph for an example program and the flow graph for a function in this program that contains an OpenMP parallel loop. After outlining, the loop is moved into a newly created function, and the call graph has an extra node. We apply machine-independent optimizations to these outlined functions and then later, during interprocedural analysis in the link step, we clone them. We now have two copies of parallel outlined functions, one for the PPE and one for the SPE, and we can optimize them independently. When cloning outlined parallel functions, we ensure that we also clone any other functions called from the outlined parallel function. Thus, in Figure 6, all functions in the subgraph rooted at the outlined function are cloned. The call sites within SPE functions are modified to invoke the SPE version of the receiver of the call instead of the PPE version. Also, a cloned function may be nested within another cloned function and may refer to data that belongs to the enclosing function. In such cases data references in nested SPE functions must be modified to refer to data that belongs to the enclosing SPE function. Because cloning of parallel code sections occurs in the link step, the compiler can generate versions of all library code as appropriate for both the SPE and PPE.

A runtime library enables parallel execution. This library includes functions for initialization, work distribution, and synchronization of data as well as control flow. The compiler inserts calls to runtime library functions appropriate to the OpenMP directives contained in the code. The OpenMP master thread runs on the PPE processor and uses the runtime library to distribute work to SPE processors. The master thread itself partakes in all work-sharing constructs. Because there is no operating system support on the SPE, this thread also handles all operating-system service requests. The PPE runtime library includes the facility to create new SPE threads and terminate them. When a new SPE thread is created, it continuously loops, waiting for

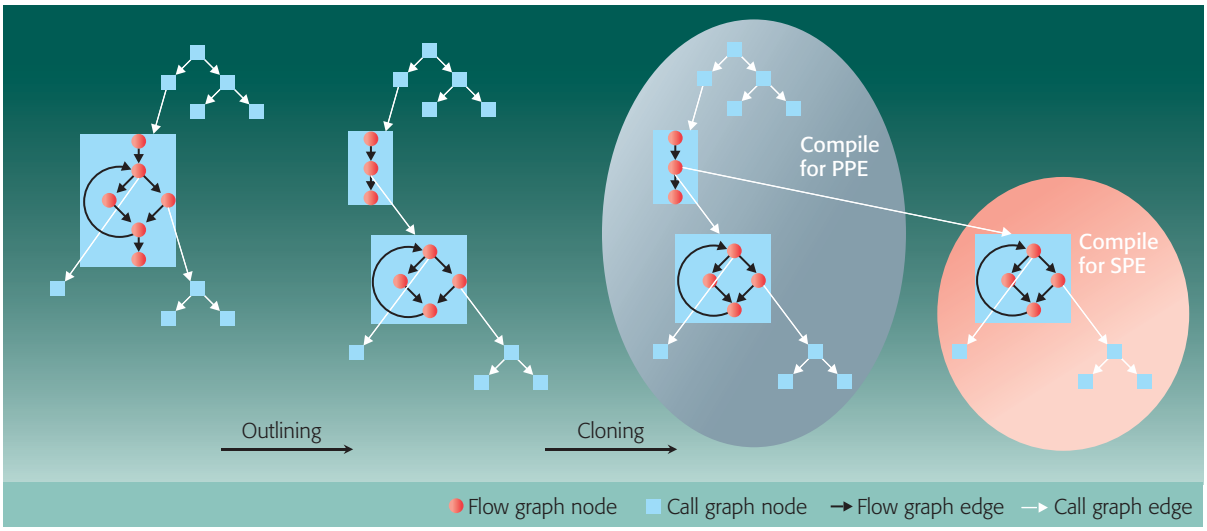


**Figure 6**
Parallelization process

the PPE to assign work items to it for execution. A work item specifies a handle that determines the function to execute, any input parameters for the function, and if need be, the current address of the stack in system memory.

The runtime library requires communication between the PPE and the SPEs for the coordination of execution. SPEs use explicit DMA to read work items assigned to them from a circular queue that is shared with the PPE. The Cell BE architecture also includes efficient communication channels in the form of signal registers and mailbox queues. The PPE uses asynchronous signals to inform an SPE that work is available or that it should terminate. The SPEs use the mailbox to update the PPE on the status of their execution.

When an SPE executes a work item, it must use DMA to access any reference to shared memory. For this purpose, the compiler identifies all shared-memory references in the code to be executed on an SPE. In the following subsection, we describe how our system automatically handles DMA transfers to and from shared memory, thus providing a single shared-memory abstraction.

### Single shared-memory abstraction

It is usual for a programmer to view a computer system as possessing a single addressable memory and for all the program data to reside in this space. In the Cell BE processor, the local stores, which alone are directly addressable by their respective SPEs, are memories separate from the vastly larger system memory. Each SPE can transfer data, by means of the DMA engine, between its local store and system memory. In our approach we attempt to abstract the concept of separate memories by allocating SPE program data in system memory and having the compiler automatically manage the movement of this data between its home location and a temporary location in the local store. A naïve compiler inserts an explicit DMA transfer for each access to shared memory, which is likely to debilitate performance. Our compiler employs a software cache mechanism that permits reuse of the temporary buffers in the local store, so that a DMA transfer is not needed for each access to shared memory. Moreover, there are many ways that the compiler can optimize these data transfers, especially when memory references are regular.

### Compiler-controlled software cache

When compiling SPE code, the compiler identifies data references in system memory that have not been optimized by using explicit DMA transfers and inserts code to invoke the software-cache mechanism before each such reference. Our current implementation provides a 4-way associative cache, and all four ways are probed inline (i.e., each set in the cache is searched simultaneously without calling a subroutine), exploiting the SIMD parallelism of the instruction set.

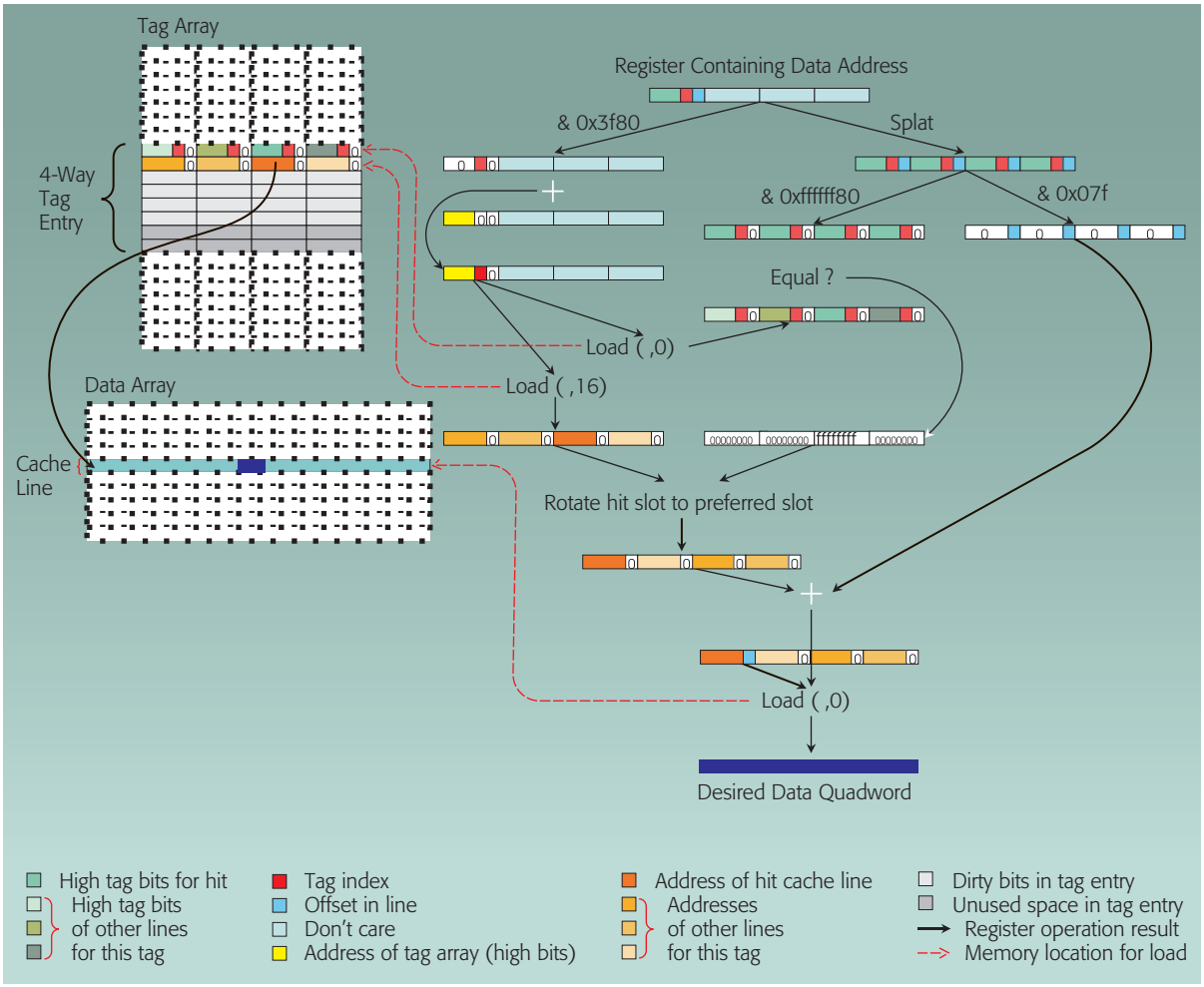


**Figure 7**
Hit logic in the software cache

*Figure 7* illustrates the cache lookup. Starting with the address of the desired variable in the primary slot of an SPE register, instructions are executed to produce the offset address of the appropriate directory entry in the tag array (a tag is a label, which in this case is part of the address and is used as a comparand to determine if the data is already in the cache). This task consists of masking all bits in the address except for those that are used to index into the tag array. To this offset we add the base address of the tag array, and using this address, we load two consecutive quadwords from the array. If any of the ways contains the tag we are seeking, then the 128-bit result of the comparison is nonzero, and we can use this to test for a cache hit. If the result is zero, this indicates a miss, and the miss handler is invoked.

Because the call to the miss handler is not expanded by the compiler until very late in the compilation, it does not appear to the optimization as a call. This allows a hit (which we assume to be the common case) to incur no penalty due to the call setup. We add the line offset to the line address and load the desired data. Store processing proceeds in much the same way, but additional instructions are required to set the "dirty bits," that is, bits which indicate that data in the cache has been modified. It takes roughly 12 instructions to process a cache hit, and a similar number to set dirty bits for a store, but because some of these instructions are dual-issued and there is often other independent work that can be scheduled, the cost in cycles is not so high; nevertheless, it is clearly very important to attempt

to reduce the number of cache lookups inserted when compiling a program.

Our cache implementation can be used for either a serial or a parallel program. There is some additional cost involved in supporting parallel execution, because we must keep track of which bytes in a cache line have been modified in order to support multiple writers of the same cache line. If the compiler knows that no data will be shared between SPEs (for example, in the case of a serial SPE application), the additional cost of maintaining these dirty bits can be avoided.

Because the compiler must deal with the potential for aliases between cached data and data obtained by other means (e.g., explicit prefetch), the miss handler must take more care in choosing a cache entry to be evicted than is normally the case with a hardware implementation.

### Data transfer optimizations

Replacing each load and store operation with the longer instruction sequence required for cache lookup may significantly increase program execution time. As a result, our memory abstraction scheme is most effective when we can minimize cache usage. To increase performance, we strive to avoid cache misses, minimize cache lookups, and optimize cache-lookup code sequences.

In general, scalar data and small structured variables do not represent a large fraction of the space requirements of a program. If we can determine that they are not used to communicate between parallel tasks, we can directly allocate them to the SPE local stores. The local, stack-allocated variables of a function often belong to this class. The analysis that determines whether these variables are not shared by multiple threads is complicated by the fact that pointer usage may cause aliasing between local and global variables. Also, large local arrays cannot be optimized in this manner, as there may be insufficient space to allocate them in the SPE local store.

Another possible optimization (which we have not yet implemented) is to explicitly allocate space for small shared variables in each local store as well as in system memory. The system memory space is the home location that contains the latest values available to all processors. The compiler can then insert explicit DMA operations to prefetch these variables into the local stores and write them back to the home location in accordance with the memory consistency model.

For large arrays, in some cases it is possible to use well-known program-restructuring techniques[15,16] such as loop blocking to allow multiple elements to be explicitly fetched in a single operation. Further restructuring can effectively "software pipeline" the blocking loop, so that data movement and computation are overlapped, essentially prefetching the data. As is the case when compiling for more traditional memory hierarchies, the compiler can perform tiling to increase locality.[17,18] Tiling uses loop-restructuring transformations such as blocking and interchange, but these transformations may not always be possible. Our compiler uses prefetching and tiling in conjunction with explicit DMA transfers. As a result, the original variables are replaced by references to much smaller temporary arrays, and this implies the rewriting of the indexing expressions as well. Although tiling and prefetching techniques have been previously used to improve locality for cache memories, their use in the context of the SPE software cache is somewhat different. In our compiler, the software-cache mechanism needs to be aware of the explicit DMA transfers for tiled data and exercise certain constraints when caching related storage locations.

For many of the cases mentioned here, a further optimization involves combining data transfers by using DMA list commands, or (in the case of multiple contiguous requests) by simply fusing several DMA operations into one. This has the effect of reducing the setup cost for DMA operations. To increase the opportunities for combining DMA operations, the compiler can reorder variables with respect to each other, including breaking up or combining structures, at link time. Our compiler already performs such transformations for other reasons, but we have not yet explored this additional use of that optimization.

For all of the foregoing techniques, when they can be used, the need for cache lookup is eliminated in effect. Ultimately, it should be necessary to use the cache only to ensure correctness for references that are irregular or that are unknown at compile time. Clearly, to the extent that this can be accomplished, substantial compile-time analysis is required. Our work in this area is still in progress, but we see very

encouraging results from our implementation thus far.

## Code partitioning

Because the limited-size local memory of the SPE must accommodate both code and data, there is always the possibility that a single SPE object will be too large to fit. We propose a code partitioning technique to reduce the impact of the local store limitations of the SPE on the program code segment. This technique can be used in a stand-alone manner with the SPE compiler or to manually partition applications. For OpenMP single-source compilation, our code partitioning is integrated with the data software cache to allow large outlined functions with large data to run seamlessly across multiple SPEs.

In our code partitioning approach, the SPE program is divided into multiple partitions by the compiler. Currently, the basic unit of partitioning is a function. The home locations of code partitions are in system memory. These SPE code partitions are overlaid during linking, so that they are all assigned the same starting virtual address, and the SPE code then fits into a virtual address space equal to the size of the largest code partition. Thus, we can use the partition size to control the space used in the local store for code.

Overlaid partitions cannot run at the same time. This implies that if code in one partition calls a function in another partition, the two partitions need to be swapped in and out of the local store at the point of the function call and return. To run a partitioned program, such partition transitions must be handled properly, and this is done collaboratively by the compiler and the runtime partition manager.

### Runtime partition manager

When the compiler partitions SPE code, it also reserves a small portion of the SPE local store for the partition manager. The reserved memory is divided into two segments: one to hold the continuously resident partition manager and the other to hold the current active code partition. The code partitions need to be relocatable, which implies that function calls should not use absolute addressing. The partition manager is responsible for loading partitions from their home location in system memory into the local store during an interpartition function call or an interpartition return. The compiler modifies the original SPE program to replace each interpartition call with a call to the partition manager. Thus, the partition manager is able to gain control and handle the transition from the current partition to the target partition. It also modifies the return address on the stack before branching to the called function to ensure that control returns to the partition manager first.

When an interpartition call is directed through the partition manager, a function pointer and arguments to the function are passed to the partition manager. The partition manager must determine which partition contains the called function. For this purpose, the compiler assigns an index to all partitions that it creates, and encodes the corresponding index in the function pointer that is passed to the partition manager. An SPU pointer is 32 bits, but because the local storage is 256 KB, only 18 bits are used. Of the 14 unused bits, 13 bits are used for the partition index (the most significant bit indicates special handling for calls to certain library functions). For example, an interpartition call to function *foo* in partition 3 is transformed from *foo (arg1, arg2, . . .)* to *call_partition_manager (3 << 18 | foo, arg1, arg2, . . .)*. The partition manager uses the partition index to fetch the correct partition and transfers control to the proper location within this partition by using the lower 18 bits of the function pointer.

*Figure 8* shows the processing steps in the execution of an automatically partitioned program. The program begins execution on the PPE and invokes code to create SPE threads. Each SPE thread is created by use of a generic SPE driver, which is an SPE binary embedded as data in the PPE object. During initialization, the partition index table, the data section for SPE data, and the first SPE code partition are copied from system memory to appropriate locations in the local store. The first SPE code partition contains the entry point function for the user SPE code, and the partition manager passes control to this function. Thereafter, the partition manager is invoked on each interpartition call, and it transfers the correct partition from system memory to local store, passes control to the correct function within that partition, and ensures that on return, control is passed back to the point after the call site in the original partition.

### Compiler transformations for code partitioning

In our implementation, we modify the interface for functions that are invoked by interpartition calls to

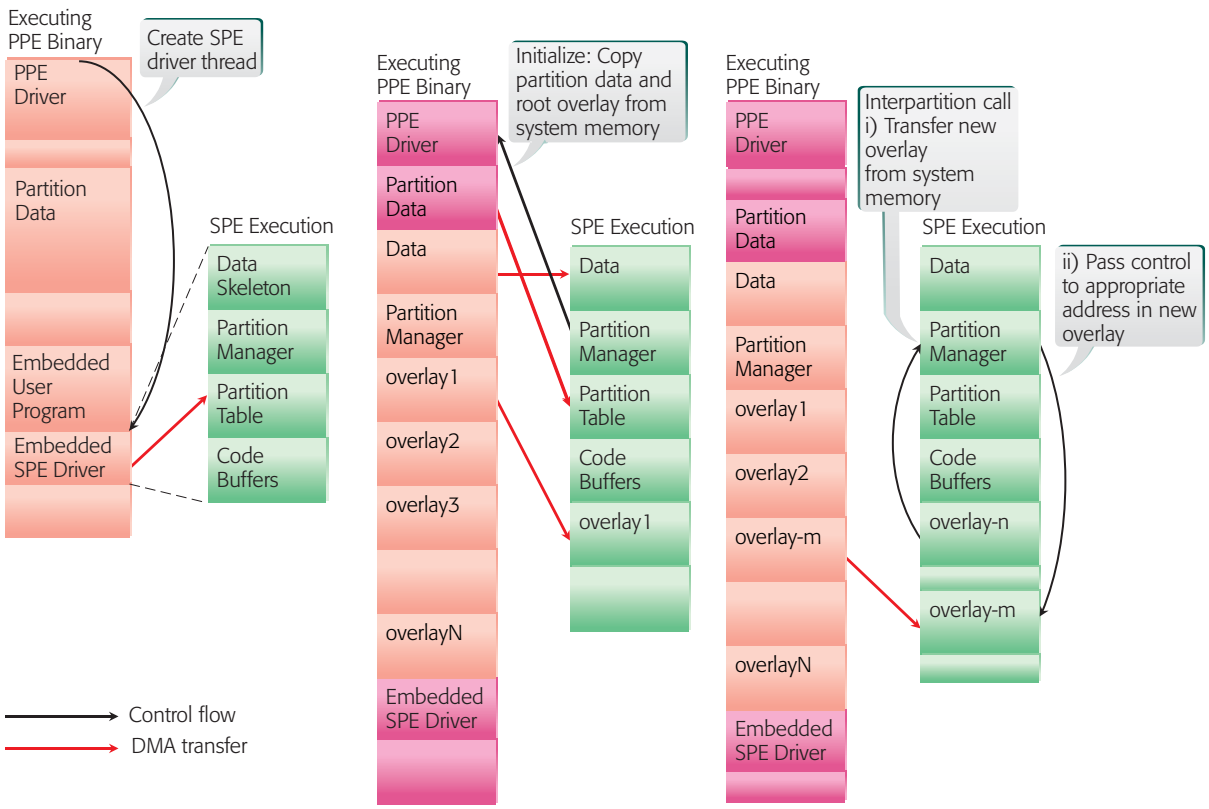


**Figure 8**
Execution of an automatically partitioned program

include an extra dummy parameter. When the compiler transforms interpartition calls, it need only change the name of the function being invoked and set the value of the dummy parameter to be the function pointer. This also enables the runtime partition manager to efficiently invoke the target function, as the arguments for the call are already in the correct locations based on linkage conventions.

The partitioning algorithm is based on a call graph. Because individual functions are normally small enough to fit into the SPE local store, this approach enables correct partitioning in most cases. Occasionally we encounter large single functions, mostly due to aggressive optimizations such as function inlining and loop unrolling. For SPE code, these optimizations constrain the space available for data in the local store and may result in additional DMA operations that adversely impact performance. Because of this, we avoid excessive inlining and loop unrolling for SPE code. There are rare instances in which functions are too large even before optimizations. In such cases, outlining can be applied to a portion of the original function, reducing the size of any single function.

*Figure 9* illustrates how the partitioning algorithm works on an example call graph where the size of each function (shown inside the circle) is assumed to be 300, and the limit on the size of a partition (shown as an oval) is assumed to be 1000. The result is two partitions with three functions in each partition, and the estimated number of interpartition calls is minimized to 150 (calls are shown as labels on the graph edges).

The schema in *Figure 10* shows the process of compiling and linking an executable that uses SPE code partitioning. The SPE XL compiler, called "spuxlc," first compiles SPE source code to object files. The interprocedural link phase then performs partitioning and generates multiple object files, one for each overlaid code partition. It also generates an object file containing a data section that defines all global data. The SPE linker is then used to produce two different SPE binaries: the generic SPE driver

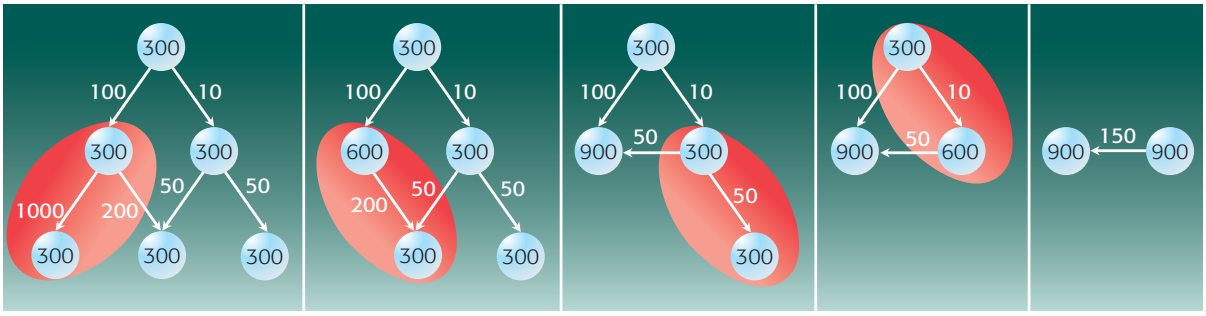


**Figure 9**
Partitioning the call graph

and the user program. The SPE driver represents the SPE binary layout during program execution. It includes space for a data section, a code section that contains the runtime partition manager code, the partition table that identifies partitions based on their index, and a code buffer that will hold one or more user SPE code partitions. The user program binary corresponds with the SPE driver binary in that the data section and the runtime partition manager code (including the partition table) are the same size and are bound to the same starting virtual addresses. This script also determines how much space to allocate for the data section and the partition table when generating the SPE driver binary. An embedder is used to wrap the SPE binaries as data sections within PPE-format object files. These object files are then linked in with the PPE code, and they exist as data sections in the final PPE user executable.

### *Optimizations for code partitioning*

With code partitioning enabled in stand-alone mode, performance is fair when executing partitioned functions on a single SPE relative to execution on the PPE. Given the preliminary nature of this work, these results are encouraging.

There are several opportunities that we are currently exploring to improve the overall performance of our code partitioning algorithm. To achieve the best results, profiling can be used instead of static estimation. Also, using the actual partition size rather than the size conservatively estimated in the compiler can improve the utilization of the local code buffer significantly. The accurate size of partitions can be determined if the size of each function is known. During a first pass, minute partitioning is performed, and each function is placed in a separate partition. The user binary is generated, and the size of each partition (each function) is extracted and saved for later use. In the second pass, code partitioning under the actual buffer size limit is performed, using accurate size information for each function.

The most promising optimization is to anticipate the next interpartition call and prefetch the corresponding code partition. This has the potential to hide the latency incurred when fetching partitions from system memory. However, this optimization requires multiple buffers, implying a much smaller partition size limit. The net effect on performance will depend on the prefetching algorithm and the accuracy of the cost model applied.

## MEASUREMENTS

We first evaluate the optimized SPE code generation techniques presented in the section "Optimized SPE code generation." *Figure 11* presents the reduction in program-execution time for each optimization relative to the performance of the original compiler. We achieved a reduction which ranged from 11 to 51 percent, averaging at 22 percent.

The benchmark programs used here are highly optimized, SIMDized kernels representative of typical workloads running on the SPEs. Kernels include a variable length decoding (VLD) from MPEG (Motion Picture Experts Group) decoding, a Huffman compression and decompression, an IDEA (International Data Encryption Algorithm) encryption, an "LU" (lower/upper triangular matrix decomposition), and a ray tracing (OnerayXY). Also included are numerical kernels such as an FFT (fast Fourier transform), a

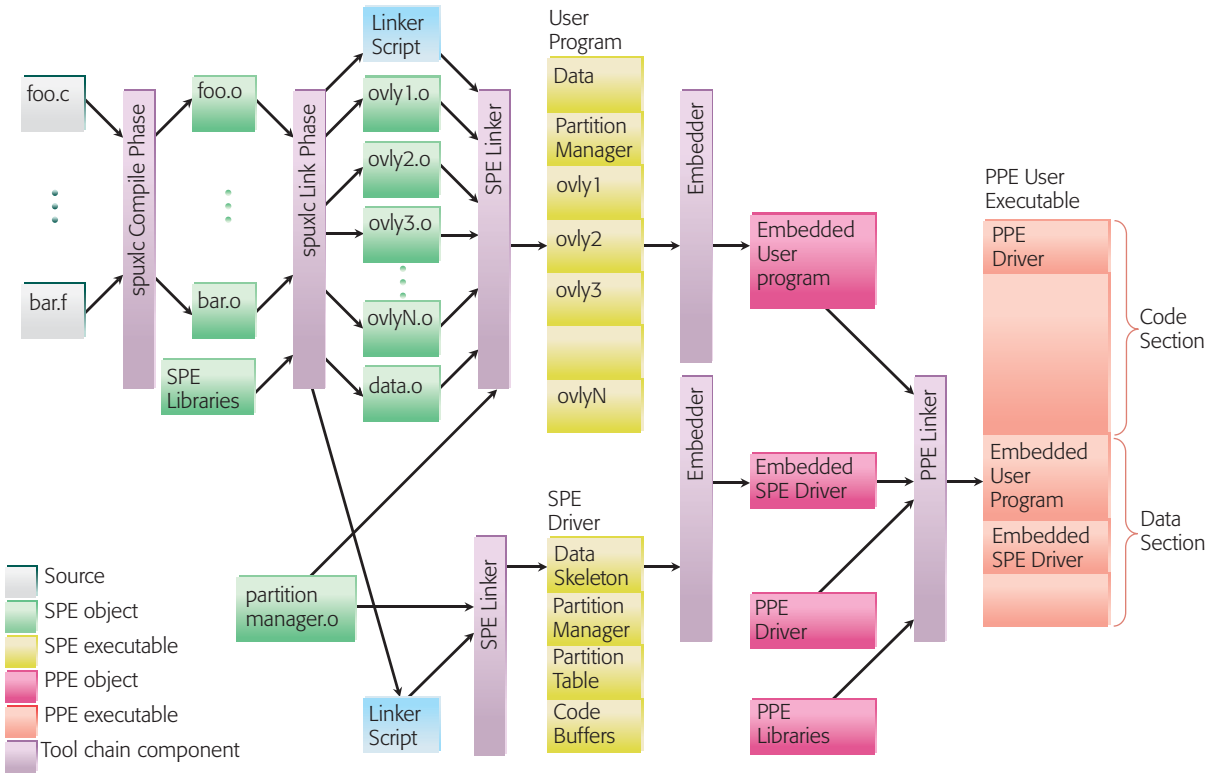


**Figure 10**
Compiling and binding an automatically partitioned program

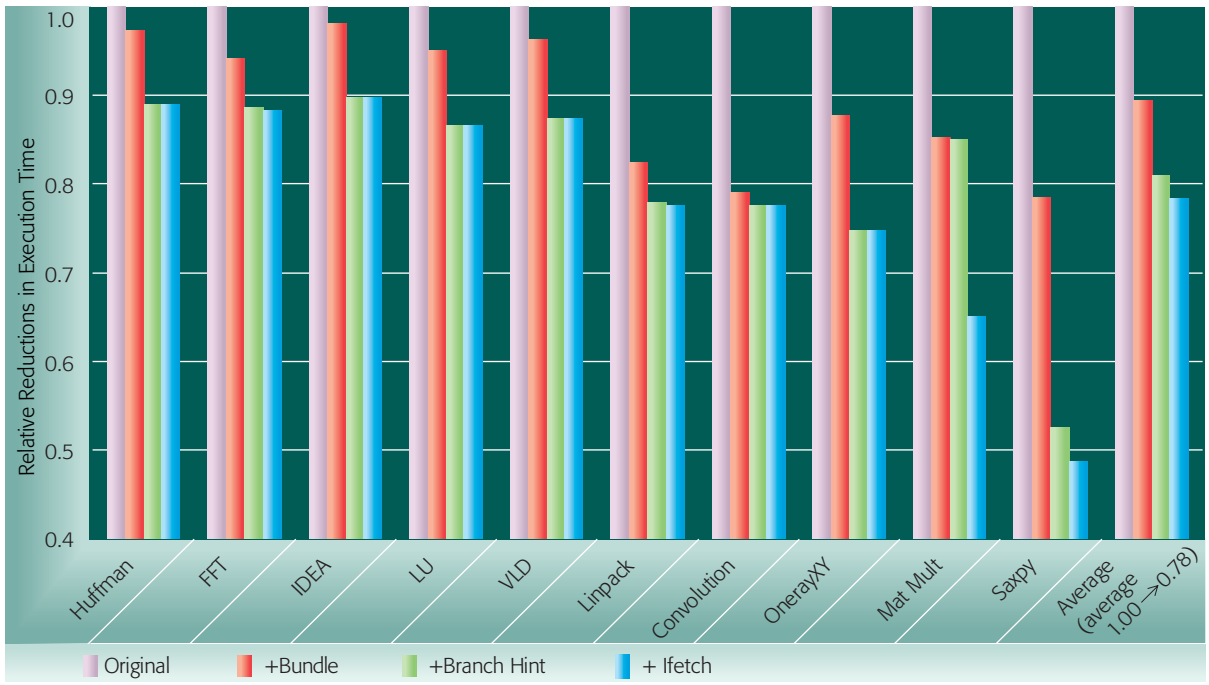


**Figure 11**
Reduction in program execution time with optimizations

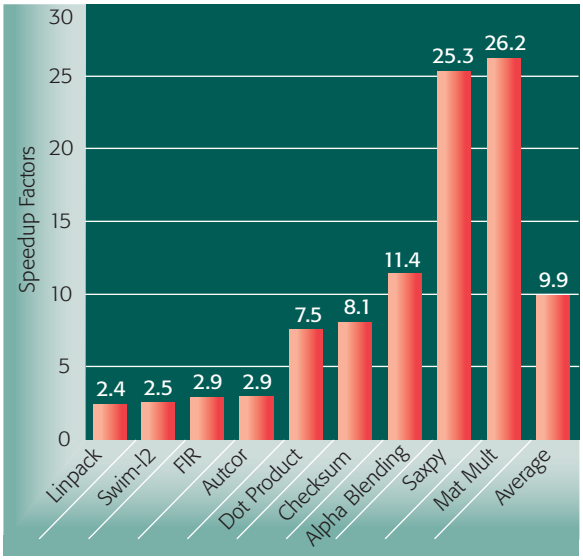


**Figure 12**
Speedup factors for auto-SIMDization

$7 \times 7$ short integer convolution, and a $64 \times 64$ float matrix multiplication.

Bundling for dual issue results in an execution-time reduction of from 2 to 22 percent, averaging at 11 percent. Large reduction percentages indicate benchmarks with large amounts of instruction-level parallelism and no "lucky" instruction alignment (where random instruction layout does not satisfy the dual-issue constraint).

Hinting predictable branches results in a further execution-time reduction of from 0 to 26 percent, averaging at 9 percent. Large reduction percentages indicate predictable branches with a sufficient amount of work to hide the hint latency. Some of the small reduction percentages (such as 0 percent for matrix multiplication) indicate such tight loops that hinting is not beneficial without jointly addressing instruction starvation.

Generating explicit instruction fetches results in a further 2 percent average execution-time reduction, with peak impact for very tight loops such as the 20 percent reduction for matrix multiplication. Subword optimization results in a further 1 percent average reduction, with peak impact of 3 percent for kernels including short type computations.

For the auto-SIMDization techniques presented in the section "Programming for the Cell BE architecture," ***Figure 12*** presents the speedup factors achieved when automatically SIMDizing sequential code kernels. Comparisons are performed at the same level of optimization, including high-level, interprocedural optimizations in addition to all of the SPE optimizations presented in the section "Optimized SPE code generation." We report an average speedup factor ranging from 2.4 to 26.2, averaging at 9.9. The benchmark programs include video, numerical, and telecommunication applications. Kernels include a short integer finite impulse response (FIR), an auto-correlation kernel, an integer dot product, a TCP/IP checksum, a "Saxpy" (i.e., a short-precision computation of `ax + y`), a matrix multiplication, and a solver kernel for Linpack, a collection of Fortran subroutines that analyze and solve linear equations and linear least-squares problems.

There are two tiers of benchmarks. The four leftmost kernels in Figure 12 achieve speedups which are fair (2.4 to 2.9) but below average. The rightmost five kernels get significant speedup (7.5 to 26.2). Both the dot product and checksum kernels performed a reduction which is not natively supported by the SPE's instruction set. This introduced some overhead which, in these two cases, can be efficiently hidden by using partial-sum reductions.

***Figure 13*** shows the effects of the parallelization discussed in the section "Generation of parallel code," presenting the results for parallel execution using only the software cache. We show results from the Spec OMP2001 suite of benchmarks (shown in the figure as apsi, ammp, applu, art, equake, Mgrid, Swim, and wupwise). We find that we achieve a speedup factor of greater than 2.5 on three of the eight benchmarks. Considering that the software cache is essentially a "fallback" strategy for code with irregular data accesses and that we have not yet fully tuned it for performance, we consider this to be a very encouraging result.

Some comparisons of our software cache and optimized data transfer results can be seen in ***Figure 14***. In this figure, calc1, calc2, and calc3 are kernels from the program Swim, whereas resid, psinv, and rprj3 are kernels from the program Mgrid. On average, a speedup factor of approximately 3 is obtained with the software cache. When data transfer optimizations are applied, the average speedup factor improves to approximately 8. The baseline execution in this case is running on a single

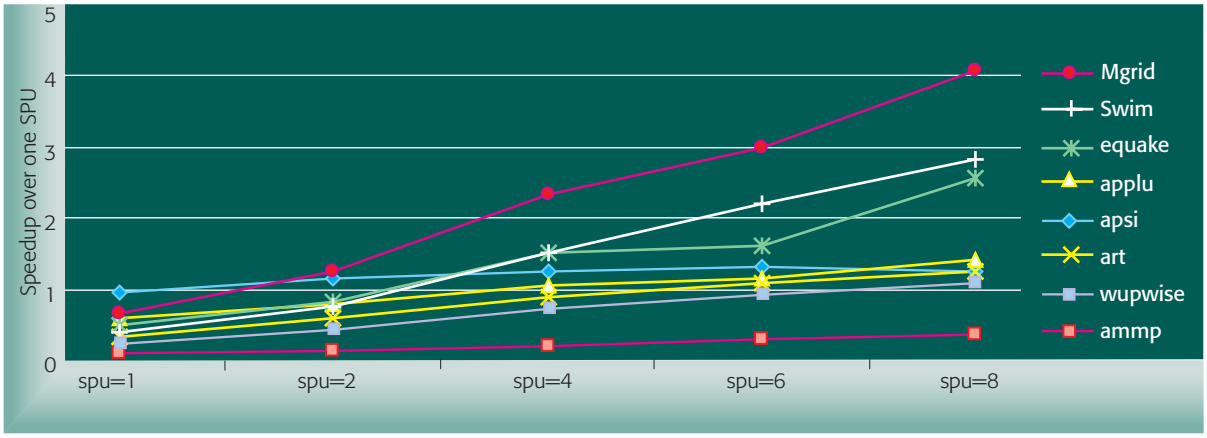


**Figure 13**
Speedup resulting from parallelization

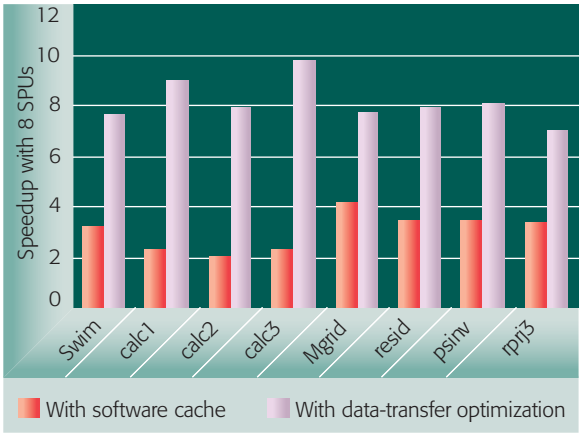


**Figure 14**
Impact of optimization on Swim, Mgrid, and their kernels

PPE, and the speedups are obtained when running on eight SPUs. We observe that optimization improves the performance by eliminating the cache lookup overhead, using more precise information about the data accesses to fetch larger chunks of data per DMA access and overlapping computation with the DMA access. The data transfer optimization work is ongoing. Because this work will allow us to eliminate or minimize cache accesses, we expect to see increased performance improvements as we make progress in this area.

## CONCLUSIONS AND FUTURE WORK

The key to unleashing the performance potential of the powerful new Cell BE architecture is the exploitation of parallelism at various levels of the system. Highly skilled programmers can achieve this with manual techniques, but sophisticated compiler technology enables significant performance potential for a broader community, resulting in a much higher rate of productivity. In this paper, we have presented, in the context of a research prototype, several compiler techniques that aim at automatically generating high-quality code by using the wide range of heterogeneous parallelism available on the Cell BE processor.

Our Cell BE compiler implements SPE-specific optimizations, including support for compiler-assisted memory realignment, branch prediction, and instruction fetching. It addresses fine-grained SIMD parallelization as well as more general OpenMP task-level parallelization, presenting the user with a single shared-memory image through compiler-mediated partitioning of code and data and the automatic orchestration of the data movement implied by this partitioning. Using benchmarks suitable to this platform, we demonstrate average speedup factors of 1.3 for SPE-specific optimizations, 9.9 for SIMDization, and 6.8 for task-level parallelization.

We are working on integrating and refining current techniques and further exploiting opportunities available on the Cell BE architecture for our target workloads. Ultimately, producing optimal code for the Cell BE processor depends on fine tuning a number of heuristics and developing an economic model that takes account of the various complexities

of the architecture. Well-known cost models for optimizing the overlap of data communication and computation are being incorporated into our approach. Other parameters for optimization can be incorporated into our model, given the heterogeneous nature of the architecture[19,20] and the particular characteristics of the SPE.

Other factors to be addressed by the model include the partitioning of memory between user code and data, resident helper code, and the cache directories. Unlike a hardware cache, our compiler-managed approach affords a degree of flexibility in modeling the size and type of the cache at compile time based on the nature of the user code. We plan to conduct extensive analysis of the usage and occupancy of our compiler-controlled cache as input to this work.

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both.

## CITED REFERENCES


1. D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The Design and Implementation of a First-Generation CELL Processor," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference (ISSCC 2005)* IEEE International, Piscataway, NJ (February 2005), pp. 184–185, http://www-03.ibm.com/industries/telecom/doc/content/bin/tc_isscc_10.2_cell_design.pdf.
2. *PowerPC Microprocessor Family: AltiVec Technology Programming Environments Manual,* IBM Corporation (July 2004).
3. J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell Multiprocessor," *IBM Journal of Research and Development* **49**, No. 4/5, 589–604 (July/September 2005).
4. S. Larsen and S. Amarasinghe, "Exploiting Superword-Level Parallelism with Multimedia Instruction Sets," *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, ACM Press, New York (June 2000), pp. 145–156, http://portal.acm.org/citation.cfm?id=349320.
5. J. Shin, M. Hall, and J. Chame, "Superword-Level Parallelism in the Presence of Control Flow," *Proceedings of the International Symposium on Code Generation and Optimization* (March 2005), pp. 165–175, http://doi.ieeecomputersociety.org/10.1109/CGO.2005.33.
6. Aart Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian, "Automatic Intra-Register Vectorization for the Intel Architecture," *International Journal of Parallel Programming* **30**, No. 2, pp. 65–98 (April 2002).
7. D. Naishlos, M. Biberstein, S. Ben-David, and A. Zaks, "Vectorizing for a SIMDD DSP Architecture," *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems* (October 2003), pp. 2–11.
8. Crescent Bay Software - VAST/AltiVec, http://www.crescentbaysoftware.com/vast_altivec.html.
9. N. Sreraman and R. Govindarajan, "A Vectorizing Compiler for Multimedia Extensions," *International Journal of Parallel Programming* **28**, No. 4, 363–400 (August 2000).
10. C. G. Lee and M. G. Stoodley, "Simple Vector Microprocessors for Multimedia Applications," *Proceedings of the 31st International Symposium on Microarchitecture*, IEEE Computer Society Press, Los Alamitos, CA (1998), pp. 25–36, http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=290951.
11. A. E. Eichenberger, P. Wu, and K. O'Brien, "Vectorization for SIMD Architectures with Alignment Constraints," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM Press, New York (June 2004), pp. 82–93.
12. P. Wu, A. E. Eichenberger, and A. Wang, "Efficient SIMD Code Generation for Runtime Alignment and Length Conversion," *Proceedings of the International Symposium on Code Generation and Optimization*, IEEE Computer Society Press, Los Alamitos, CA (March 2005), pp. 153–164.
13. P. Wu, A. E. Eichenberger, A. Wang, and P. Zhao, "An Integrated SIMDization Framework Using Virtual Vectors," *Proceedings of the 19th Annual International Conference on Supercomputing,* ACM Press, New York (June 2005), pp. 169–178.
14. *Official OpenMP Specifications,* OpenMP Architecture Review Board (2002), http://www.openmp.org/specs/.
15. T. C. Mowry, "Tolerating Latency through Software-Controlled Data Prefetching," Doctoral dissertation, Stanford University (March 1994).
16. M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm," *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, ACM Press, New York (May 1991), pp. 30–44, http://portal.acm.org/citation.cfm?id=113449&coll=Portal&dl=GUIDE&CFID=54819031&CFTOKEN=14228294.
17. G. Rivera and C.-W. Tseng, "Tiling Optimizations for 3D Scientific Computation," *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, IEEE Computer Society, Washington, DC, Online proceedings (November 2000) http://portal.acm.org/citation.cfm?id=370403&coll=Portal&dl=GUIDE&CFID=54819031&CFTOKEN=14228294.
18. A. Badaway, A. Aggarwal, D. Yeung, and C.-W. Tseng, "Evaluating the Impact of Memory System Performance on Software Prefetching and Locality Optimizations," *Proceedings of the 15th International Conference on Supercomputing*, ACM Press, New York (June 2001), pp. 486–500, http://portal.acm.org/citation.cfm?id=377906&coll=Portal&dl=GUIDE&CFID=54819031&CFTOKEN=14228294.
19. J. Andrews and C. Polychronopoulos, "An Analytical Approach to Performance/Cost Modeling of Parallel Computers," *Journal of Parallel and Distributed Computing* **12**, No. 4, 343–356 (August 1991).
20. D. J. Lilja, "A Multiprocessor Architecture Combining Fine-Grained and Coarse-Grained Parallelism Strategies," *Journal of Parallel Computing* **20**, No. 5, 729–751 (May 1994).

*Accepted for publication September 21, 2005.*
*Published online January 19, 2006.*

*Alexandre E. Eichenberger*
*IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (alexe@us.ibm.com).*
Dr. Eichenberger is a research staff member in the Exploratory System Architecture department at the Watson Research Center. He received a diploma in computer science from Eidgenössische Technische Hochschule in Zurich, Switzerland in 1991, and M.S. and Ph.D. degrees in computer and electrical engineering from the University of Michigan at Ann Arbor in 1993 and 1996, respectively. He was a faculty member of the Department of Electrical and Computer Engineering at North Carolina State University before joining IBM. In addition to his current work in auto-SIMDization, his research interests include instruction-level parallelism, predicated execution, profiling techniques, and software pipelining.

*John Kevin O'Brien*
*IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (caomhin@us.ibm.com).*
Mr. O'Brien has spent the last 24 years at IBM working in the field of compilation and architecture. Initially, at the IBM Toronto Lab, he was the architect of the TOBEY optimizing back end (used in IBM's xlc, xlf, and xlC compiler products). Since then, he has spent 17 years at IBM Research, where his research interests have included multithreaded architecture, Smalltalk, Java™, continuous optimization, binary translation and optimization, parallelization, and vectorization (including SIMDization) for several processors, most recently the Cell Broadband Engine™ processor. Mr. O'Brien received a B.Sc. degree in theoretical physics and an M.Sc. degree in astrophysics from the University of London in 1974 and 1976 respectively. Currently, he is investigating memory-related optimizations for the Cell Broadband Engine™ processor.

*Kathryn M. O'Brien*
*IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (kmob@us.ibm.com).*
Ms. O'Brien has worked at IBM for 23 years, 17 of them as a researcher at the Watson Research Center, where she has been involved in several static and dynamic compilation projects. She received a B.A. degree from Queen's University of Belfast in 1973, and an M.A. degree from the University of London in 1976. Ms. O'Brien was involved in the initial IBM XL Fortran compiler, and the early vectorization and parallelization efforts in the XL compilers. Currently, she is investigating automatic parallelization for the Cell Broadband Engine™ and other architectures.

*Peng Wu*
*IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (pengwu@us.ibm.com).*
Dr. Wu is a research staff member in the High-Performance Software Environment department at the Watson Research Center. She received M.S. and Ph.D. degrees in computer science from the University of Illinois at Urbana-Champaign in 1999 and 2001, respectively. She subsequently joined IBM at the Watson Research Center, where she has worked on compiler optimization, auto-SIMDization, and high-performance computing.

*Tong Chen*
*IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (chentong@us.ibm.com).*
Mr. Chen is a compiler research team member in the Exploratory System Architecture department at the Thomas J. Watson Research Center. He received B.S. and M.S. degrees in computer science from Fudan University, and is a Ph.D. student in computer science at the University of Minnesota. He joined IBM at the Thomas J. Watson Research Center, and his work there has focused on compilers.

*Peter H. Oden*
*IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (oden@us.ibm.com).*
Dr. Oden is a research staff member in the Systems department at the Watson Research Center. He received an A.B. degree from Columbia College in 1955, and M.S. and Ph.D. degrees in electrical engineering from Columbia University in 1958 and 1966, respectively. He joined IBM at the Watson Research Center in 1963, where he has worked on design automation, programming languages and compilers, and computer micro-architecture. He received an IBM Outstanding Contribution Award for his work on design automation in 1968 and an IBM Research Outstanding Technical Achievement Award for his work on compilers in 1981. He is an author or coauthor of several patents and technical papers.

*Daniel A. Prener*
*IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598 (prener@us.ibm.com).*
Dr. Prener is manager of the Exploratory System Architecture group in the Systems department at the Watson Research Center. He received a B.A. degree in mathematics from Swarthmore College in 1965, and M.A. and Ph.D. degrees in mathematics from the University of Pennsylvania in 1967 and 1972, respectively. Before joining IBM, he taught mathematics at the State University of New York at Stony Brook, and mathematics and computer science at Lehman College of the City University of New York. Since joining IBM in 1981, he has worked on computer architecture and compiler optimization in a variety of contexts.

*Janice C. Shepherd*
*c/o Daniel Prener, IBM Research Division, Thomas J Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (janshep@us.ibm.com).*
Ms. Shepherd is a senior software engineer and works out of her home in Grand Junction, Colorado. She received her B.S. degree from Queen's University in 1980 and her master's degree from the University of Toronto in 1983. Her current interests are in productization and multilanguage support.

*Byoungro So*
*IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (bso@us.ibm.com).*
Dr. So is a post-doctoral researcher in the Computer Architecture department at the Watson Research Center. He received a B.S. degree in computer science from Dongguk University in Seoul, Korea in 1996, and M.S. and Ph.D. degrees in computer science from the University of Southern California in 1998 and 2003, respectively. He subsequently joined IBM at the Watson Research Center, where he has worked on high-performance computing and parallelizing compilers. In 2003, he received an Outstanding Academic Achievement Award from the University of Southern California. Dr. So is a member of the Korean-American Scientists and Engineering Association.

*Zehra Sura*
*IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (zsura@us.ibm.com).*
Dr. Sura is a research staff member in the Systems department at the Watson Research Center. She received a B.E. degree in computer science from VRCE, in Nagpur, India in 1998, and M.S. and Ph.D. degrees in computer science from the

University of Illinois at Urbana-Champaign in 2001 and 2004, respectively. She subsequently joined IBM at the Watson Research Center. Her research interests include analysis and transformation of programs for parallel processing and the use of runtime techniques to improve compiler optimizations.

*Amy Wang*
*IBM Toronto Lab, 8200 Warden Ave., Markham, Ontario L6G 1C7, Canada (aktwang@ca.ibm.com).*
Ms. Wang is a member of the XL compiler back-end team. She obtained a Bachelor of Applied Science degree in 1999 and a Master of Applied Science degree in computer engineering in 2001, both from the University of Toronto. In 2002, she joined the IBM Toronto Software Lab, contributing her skills to the development of various compiler back-end optimizations, such as register allocation and most recently, auto-SIMDization for VMX hardware.

*Tao Zhang*
*College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, Georgia 30332 (zhangtao@cc.gatech.edu).* Mr. Zhang is a Ph.D. candidate in computer science at Georgia Institute of Technology. He received a B.S. degree in computer science from Peking University in 2001, and an M.S. degree in computer science from Georgia Institute of Technology in 2003. During his graduate study, he has done work with Dr. Santosh Pande on compiler and architecture optimizations for embedded systems in terms of memory cost, performance, and power. He has also been working on compiler and architecture support for system security.

*Peng Zhao*
*IBM Toronto Lab, 8200 Warden Ave., Markham, Ontario L6G 1C7, Canada (pengz@ca.ibm.com).* Mr. Zhao received a B.S. degree in computer science from Beijing University of Aeronautics and Astronautics in 1995, and an M.S. degree in computer science from McMaster University in 2000. He subsequently joined the IBM Toronto lab, where he has worked on the Toronto Portable Optimizer (TPO) team.

*Michael K. Gschwind*
*IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (mkg@us.ibm.com).* Dr. Gschwind is a research staff member at the Watson Research Center. He was one of the originators of the Cell Broadband Engine™ architecture and the architect of the SIMD-based SPU architecture. During the definition of the SPU architecture, Dr. Gschwind developed the first compiler targeting the Cell Broadband Engine architecture. He is currently leading work on future high-performance and low-power architectures focused on media, high-performance, and general-purpose computing applications. Before embarking on the Cell Broadband Engine project, Dr. Gschwind contributed to several generations of binary translation architectures exploiting instruction-level parallelism. He also contributed to the modeling and evaluation of future micro-architectures for IBM's pSeries® and zSeries® architectures. Before joining IBM in 1997, he was an Assistant Professor at the Department of Computer Engineering, Technische Universität Wien in Vienna, Austria. Dr. Gschwind received M.S. and Ph.D. degrees in computer science from Technische Universität Wien in 1991 and 1996, respectively. His research interests include compiler and computer architecture and micro-architecture, He is the author of more than 60 papers and holds numerous patents on high-performance computer architecture. Dr. Gschwind has been named an IBM Master Inventor in recognition of his technical contributions and is a Senior Member of the IEEE.?

*Roch Archambault*
*IBM Toronto Lab, 8200 Warden Avenue, Markham, Ontario L6G 1C7, Canada (archie@ca.ibm.com).* Mr. Archambault is a Senior Technical Staff Member at the IBM Toronto Lab in the compiler development area. His most significant contributions have been as an architect and technical lead in compiler-back-end and optimization technologies for IBM C, C++ and FORTRAN compiler products. He has actively participated in high-performance-computing (HPC) customer-bid situations and played an important role supporting IBM HPC marketing teams. Mr. Archambault has extensive experience in inventing and producing code in the form of prototypes or fully implemented features and is well known for his in-depth knowledge of compiler and optimization technologies.

*Yaoqing Gao*
*IBM Toronto Lab, 8200 Warden Avenue, Markham, Ontario L6G 1C7, Canada (ygao@ca.ibm.com).* Dr. Gao is a senior software engineer and is working primarily on compiler optimization. His major interests are computer architecture and compiler optimization. He received an IBM Outstanding Technical Achievement Award in 2002 and IBM Invention Achievement Awards in 2003 and 2004. Before joining IBM, he conducted research on parallel and distributed processing and programming languages at Tsinghua University, the National University of Singapore, the University of Tokyo, and the University of Alberta.

*Roland Koo*
*IBM Software Solutions Toronto Lab, 8200 Warden Avenue, Markham, Ontario L6G 1C7, Canada (rkoo@ca.ibm.com).* Mr. Koo is a Senior Manager in the compiler development area. His primary responsibilities involve managing the technical content and delivery of the Toronto Portable Optimizer (TPO), a machine-independent high-level optimizer for IBM XL C/C++ and XL Fortran compiler products. He also supports performance initiatives for delivery of new architectures for the pSeries® and zSeries® processors and research initiatives in the area of compiler optimization for future processor architectures. He joined IBM in 1989. His experience includes compiler development, product planning, and project management. He has a bachelor's degree in computer science from the University of Toronto. ■