

An introduction to compiling for the Cell Broadband Engine architecture, Part 2: Optimizing for the SPE

Think of it as an opportunity, not a challenge

Skill Level: Intermediate

[Power Architecture editors \(dwpower@us.ibm.com\)](mailto:dwpower@us.ibm.com)
developerWorks
IBM

07 Feb 2006

Second in the "[An introduction to compiling for the Cell Broadband Engine architecture](#)" series, this tutorial discusses specific issues in optimizing code to run effectively on the Synergistic Processor Elements (SPEs) in the Cell Broadband Engine™ (Cell BE) processor.

Section 1. Before you start

About this tutorial

The SPE unit is exceptionally powerful, but has very specific requirements to unleash that power. The SPE is single instruction, multiple data (SIMD) only, so all scalars have to be presented as vectors, and all vectors have to be aligned. The local store memory is single-port, and extreme data throughput can lead to instruction starvation without care to regularly fill the instruction buffer. Without branch prediction in Cell BE architecture, hints become critical to keeping the instruction buffer properly filled. Optimizing code for the even/odd dual-issue instruction logic is another performance tool. This tutorial discusses these issues and ancillary ones at length.

Prerequisites

This article presumes some basic familiarity with the architecture of the Cell BE, and some basic understanding of computer architecture. Readers who managed the [previous tutorial](#) should be fine.

Section 2. SPE optimization overview

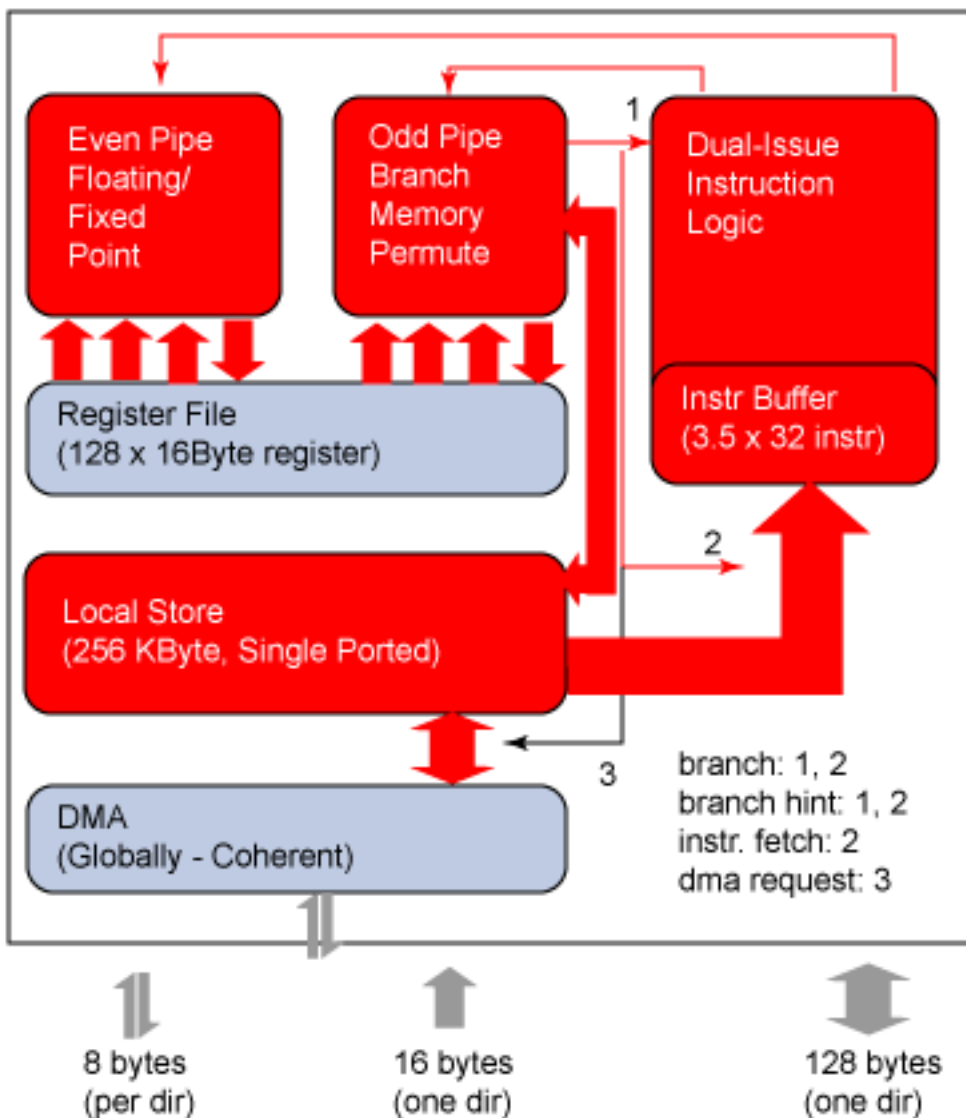
Cell BE architecture

As discussed in the [first tutorial](#) in this series, the Power Processor Element (PPE) executes traditional code. The SPEs are optimized for high computing throughput and a small area footprint. To attain this, they have low hardware complexity, and obtaining peak performance requires hardware assistance.

SPE features

This figure illustrates some of the architectural features of the SPE:

Figure 1. A block diagram of the SPE



Both of the functional units can read three registers at a time, but can only write one. One functional unit handles all arithmetic instructions; the other handles branches, memory access, and permutations.

SPE features

The functional units of the SPE have only SIMD functional units; all register and memory access is 16 bytes wide. The hardware doesn't have branch prediction, so the compiler must manage hinting and prefetching of code. This places the burden of effective branch prediction on the compiler, but dramatically simplifies the hardware on the chip. Instructions are dual-issue, but instructions must be in parallel, and properly aligned. Only one instruction can be issued to each functional

unit per cycle. Hardware does have a full dependency check, however.

Local memory is single-ported, and allows only aligned accesses. Memory contention is alleviated by the compiler, which issues load/store and DMA instructions as needed. The same functional unit handles communication between local store and the register file, and between local store and system main memory.

Section 3. Implementing scalar code on SIMD hardware

Issues in implementing scalar code

SIMD hardware imposes substantial challenges for attempts to use scalar code. The SIMD hardware used in the SPE loads data in 16-byte chunks, located at 16-byte boundaries; this data is loaded into registers, modified, and stored out to memory.

A simple operation, such as adding two values and storing the result, can face three distinct problems in execution, which are best handled by the compiler. This section discusses the addition of two values, storing the result in another location. The code under discussion represents the C expression $a[2] = b[1] + c[3]$, where a , b , and c all represent pointers with 16-byte alignment.

Problem 1: Memory alignment on load

When an address is passed to a load operation on the SPE, it is automatically truncated to a multiple of 16 bytes. The location of the desired object in the register will vary with the object's original address. If the object was precisely on a 16-byte boundary, it will be the first object in the register; if it was 12 bytes past a 16-byte boundary, it will be the last object in the register. Thus, attempts to load four consecutive objects into registers can produce the same register contents, with the desired objects in different locations. For instance, in the load of $b[1]$, the register loaded will contain $b[0]$, $b[1]$, $b[2]$, and $b[3]$, with the desired value being the second component of the vector. When $c[3]$ is loaded, it is the fourth component of the vector.

Problem 2: Alignment on operation

When adding two values loaded into registers, the addition operates in parallel on each object. If the desired items were not in the same location in each register, then

none of the resulting values will be the desired value.

For instance, in the sample expression, there is no value in the result register corresponding to $b[1]+c[3]$. The second value of the vector is $b[1] + c[1]$, and the fourth is $b[3] + c[3]$.

Problem 3: Clobbering time

The worst aspect of this, however, comes when trying to store a value. Since a whole register is being stored, the other three values within a 16-byte block are clobbered, replaced with whatever happened to be in the rest of the register. This is almost certainly not what you want! Furthermore, if the location you wish to store to isn't aligned appropriately for the register component you are trying to store there, your data will be stored in the wrong place.

Solution 1: Read and rotate

Scalar loads can be handled with a two-operation combo: a read followed by a rotate. (Both operations will be on the memory/permute functional unit.) The register is rotated so that the desired value is the first component of the vector. This solves Problem 1; it also solves Problem 2, because any two scalars loaded will be the first components of their respective vector.

Solution 2: Read-modify-write

To store data without clobbering requires a more elaborate procedure. The 16-byte chunk of memory holding the target location is loaded into another register, and a shuffle operation is performed to overwrite the target component with the calculated value. The resulting register is written out, which changes only the desired value. For instance, in the example given, the values $a[0]$ through $a[3]$ are loaded into a register, the third component of the register ($a[2]$) is replaced with the calculated value, and then the whole range ($a[0]$ through $a[3]$) is replaced. This adds up to three ops of overhead: one to load the values to be preserved, one to create the mask used by the shuffle operation (called an insertion mask), and one to shuffle the values together. In fact, the results of the mask operation and load operation may be reused later, which can reduce the effective overhead. Insertion masks are created by special instructions designed for the purpose.

Optimizations

The overhead associated with scalar operations can be dramatically reduced. In many cases, code using scalar variables can be vectorized, either by programmers

or by the compiler. Vectorized code eliminates the problems of performing scalar arithmetic on a vector processor. Another solution, which optimizes for speed at the expense of space, is to store scalar variables in the first slot of a 16-byte region, and leave the remainder of the region unused. This eliminates the need for a rotate on load, as the desired data is guaranteed to be in the first slot, and eliminates the need for a read-modify-write cycle when storing, as the other data in the chunk is garbage, and it doesn't matter whether it gets clobbered.

Section 4. Software-assisted branch architecture

The branch architecture of the SPE

The SPE has a simple branch architecture with no hardware branch prediction. It supports a single active branching hint at a time. However, predicated instructions allow some operations which are represented as branches on some processors to proceed without any branches at all.

This leaves two major options for reducing branch overhead: Using predicated instructions for small if/then/else operations, and providing hints for branches which can be predicted.

Predicated operations

Branching operations pose special challenges for vector operations, because it is common to wish to perform a given operation only on some members of a vector. For instance, given a variable a , its absolute value might be represented as $a > 0 ? a : -a$. On a scalar processor, this would be implemented as a branch. With a vector unit, you might have a vector containing both positive and negative numbers. The solution used in VMX, and also in the SPE instruction set, is to perform such operations unconditionally, then combine a mix of modified and unmodified values into a single register.

Comparison operators

Comparison operators take two vectors as operands and produce a third. The third vector has zero bits wherever the comparison was false, and one bits wherever it was true. For instance, a greater-than comparison between a vector a and a vector of all zeroes would result in a vector having zeroes in all the positions corresponding

to values less than or equal to zero, and ones in all the positions corresponding to values greater than zero.

Combining vectors

After the comparison operator has completed, and the conditional calculations have been performed, the original vector and the modified vector can be combined using the vector resulting from the comparison operator.

The selection operator takes three arguments: two vectors containing values, and one indicating which values to copy from each vector. If the comparison vector has one bits in a given element, the value is taken from one vector; otherwise, it's taken from the other. This can be used to avoid the need for branches, especially on simple operations.

Efficiency of unconditional operations

Users accustomed to scalar processors are often surprised by the apparent wastefulness of performing operations unconditionally, then discarding the results in some cases, but when this allows the processor to avoid branches, it generally makes for a substantial performance increase.

Hints for predictably taken branches

In some cases, a branch is necessary, and comparison registers and unconditional operations are a poor match for the task. For instance, function calls and returns generally require branching, and loops require branching. Some branches can be reduced or eliminated by code layout optimizations, but not all. Similarly, inlining of functions can remove branches, but may in some cases expand code too much for the comparatively small local storage of the SPE.

The HINT operation

The SPE provides a branch hint operation, which allows prefetching of branch targets. The hint instruction must be a minimum of 15 cycles before the branch will occur, and a minimum of eight operations before the branch. If the hint guesses correctly, and is scheduled early enough, it completely eliminates the branch penalty! This is another circumstance where unconditional operations which may or may not get used might pay off; if the branch hint cannot be calculated early enough, it might be productive to use it anyway and perform some calculations that might be useful.

When hints are beneficial

A hint which is often wrong is unlikely to be beneficial. A formal calculation is possible, assuming the probability of a correct guess can be estimated. The benefit of a branch hint is the probability that the branch hint would be correct, times the penalty of a branch miss (including an unhinted branch). The primary cost is the cost of the hint instruction itself, plus any overhead (for example, a cycle or two of dummy instructions to delay the branch until the hint has taken effect); furthermore, calculating the hint might include some cost. The secondary cost is the probability that the branch hint is wrong, times the penalty for a hinted branch miss.

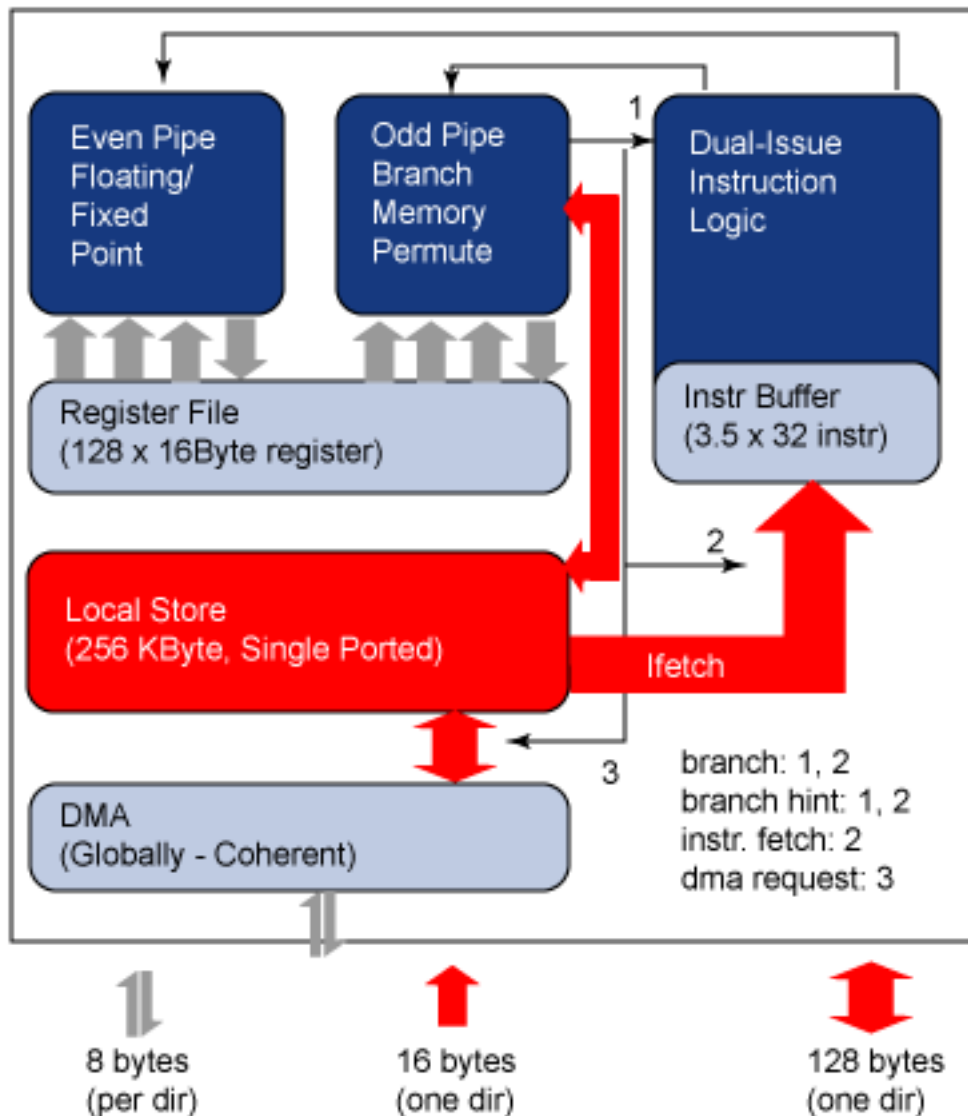
In the case where the probability of a correct prediction is high, hinting is often beneficial.

Section 5. Software-assisted instruction issue

Instruction issue restrictions

As has been mentioned in other sections, the SPE's architecture is dual-issue, but each functional unit can only be issued one instruction at a time, and their functions are non-overlapping.

Figure 2. A block diagram of the SPE



The constraints on the dual-issue architecture offer a number of challenges for the compiler. To make effective use of the SPE, the compiler must be aware of the code layout requirements. In particular, the compiler's scheduler must bundle operations accordingly.

Even/odd restrictions

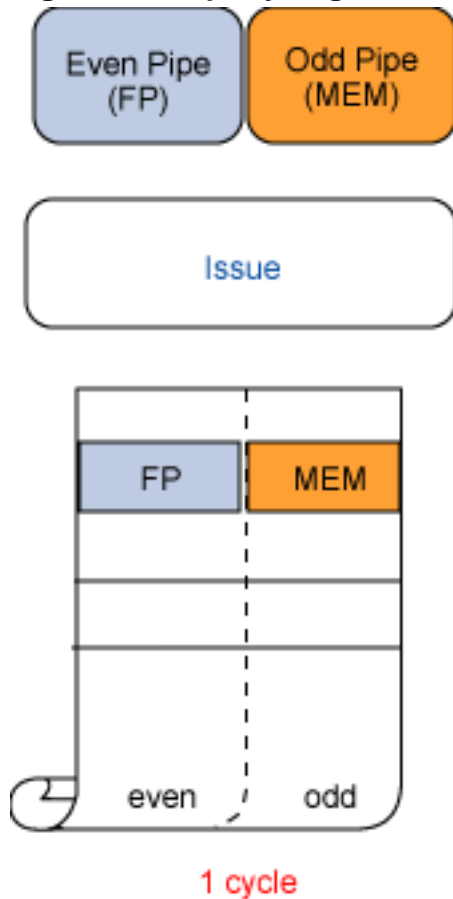
Instructions must alternate between the arithmetic and memory functional units. In this discussion, *even* addresses refer to addresses which are even-numbered multiples of instruction size, and *odd* addresses refer to addresses which are odd multiples of instruction size. Instructions at even addresses must be issued to the arithmetic functional unit, and instructions at odd addresses must be issued to the

memory functional unit. This simplifies the issuing hardware, but imposes requirements on the compiler. For instance, the compiler must know whether a given function starts on an even or odd boundary. Code which doesn't comply with these requirements will still run correctly, but noticeably slower.

The ideal case

The ideal case, starting at an even-aligned address, is an arithmetic instruction followed immediately by a memory instruction. This takes a single cycle to issue, and requires no special handling.

Figure 3. Properly aligned instructions

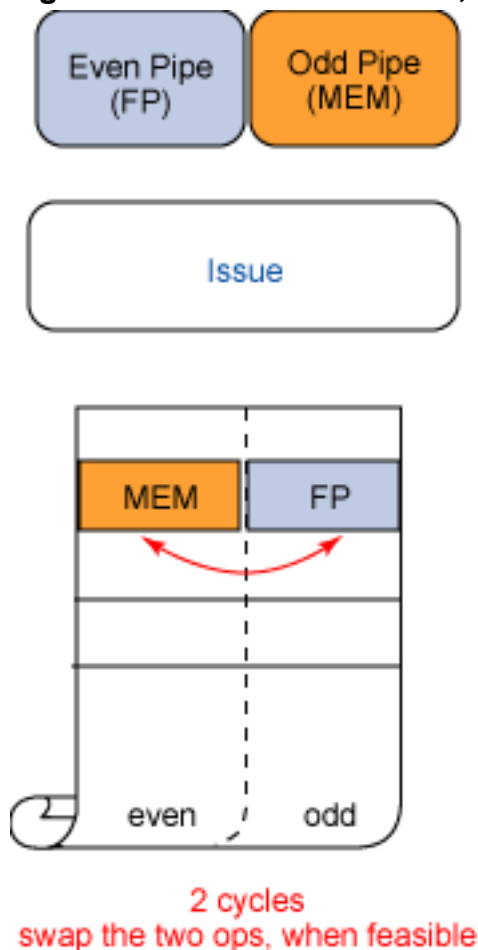


Parallel operations, misaligned

The next case to consider is when you have a memory instruction immediately followed by an arithmetic instruction. If the operators have no dependencies, they can simply be swapped. However, if their dependencies are not satisfied (the second instruction really does depend on the first), this can't be done, and a cycle of

latency will be added.

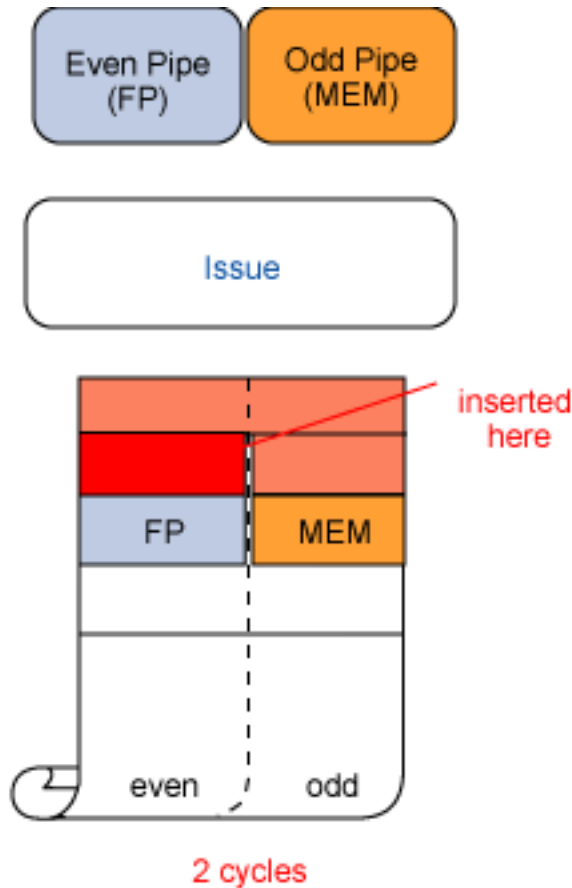
Figure 4. Parallel instructions, misaligned



Parallel operations in two consecutive issue buffers

When instructions don't come nicely paired, it is possible for instructions to be generated such that a natural pairing gets split across two issue buffers, which could set instructions up to be delivered to the wrong functional units. The scheduler tries to find the best overall scheduler, but the bundler is responsible for adding a NOP (Not an Operation) in a prior cycle to get the instruction issue back in sync. If this is successful, there will be a NOP in a previous cycle, but the new instructions are back to the ideal case. A NOP used to change code layout like this has no cost but space, and can improve performance.

Figure 5. Parallel instructions in two consecutive buffers



insert an extra op before FP to change the code layout

Alleviating instruction issue restrictions

Modifying the dependency graph to account for the latency of false dependencies (in the case of misaligned parallel operations) gives the scheduler a good chance of reordering operations to eliminate such problems. The bundler keeps track of even/odd code layout issues and performs trivial modifications (such as swapping parallel operations, or inserting NOPs as needed) to keep the code layout functional.

As mentioned above, this does require the compiler to know whether a given function starts at an even or odd code layout boundary. Instructions cannot be added after scheduling is complete, because it would impact code layout, and thus potentially violate the dual-issuing constraints. Branch hints and instruction fetches always go through the memory functional unit.

Section 6. Single-ported local memory

Local store is single ported

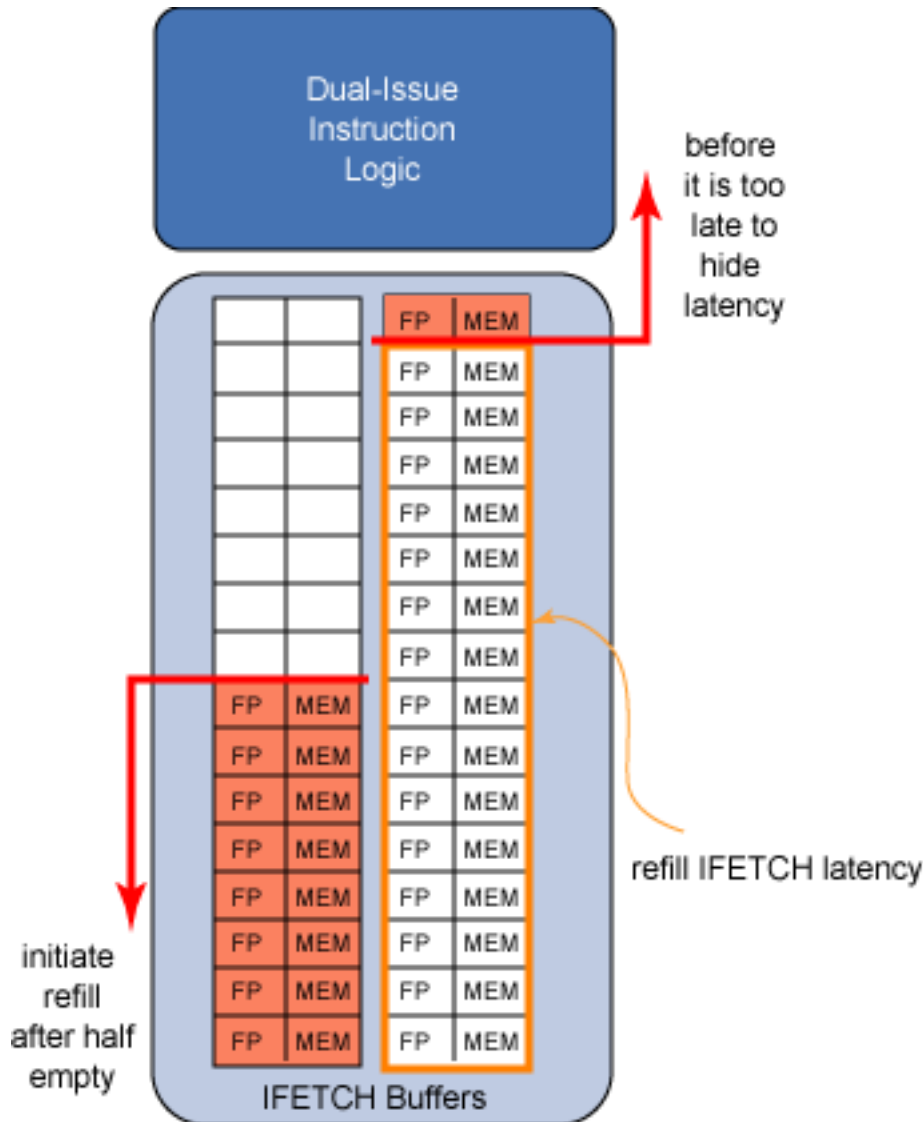
The local store on the SPE is single ported. The hardware is cheaper and simpler as a result, making it easier to get eight SPEs onto a single chip, but this imposes limitations on the compiler. The port is asymmetric, offering a 16-byte pipe to the memory functional unit (MEM), but a 128-byte pipe for DMA and instruction fetching (IFETCH).

The local store timing and priorities are fixed and deterministic. DMA has first priority, followed by MEM, followed by IFETCH. Memory accesses have priority, and it is up to software to prevent instruction starvation.

Instruction starvation

Starting with the best-case scenario (two full buffers, for a total of 64 instructions), a refill should be initiated when the first buffer is half empty. If the MEM port is continuously used, no instructions are fetched, and the buffers run out; this does prevent further use of the MEM port, but no instructions can be issued for the full duration of instruction-fetch latency. An instruction fetch only ties up memory access for a single cycle, although the latency before the instruction buffer is full is longer.

Figure 6. Instruction fetch window



Preventing instruction starvation

When the memory port is used heavily, it is necessary to initiate a refill fairly early, or the instruction-fetch latency will keep instructions from showing up in time. The window is roughly from halfway through the first buffer up to the first instruction in the second buffer; past that, there will be at least some window of instruction starvation. The latency of instruction fetches is long, so once the processor is executing commands from later in the second buffer, it will get back to the first buffer before an instruction fetch can complete.

Instruction starvation after a branch

The previous discussion assumes starting with two full IFETCH buffers. In the case of a branch, it's not so easy. If the refill is not initiated in the first cycle after the branch, the buffer will run out of instructions before the fetch is complete. Furthermore, this is true only if there's no pollution in the buffer; a hinted branch must target the first operation in the buffer. Otherwise, there is no way to prevent at least some instruction starvation. Branch hints work by filling a separate buffer, not one of the two alternating buffers the regular IFETCH uses.

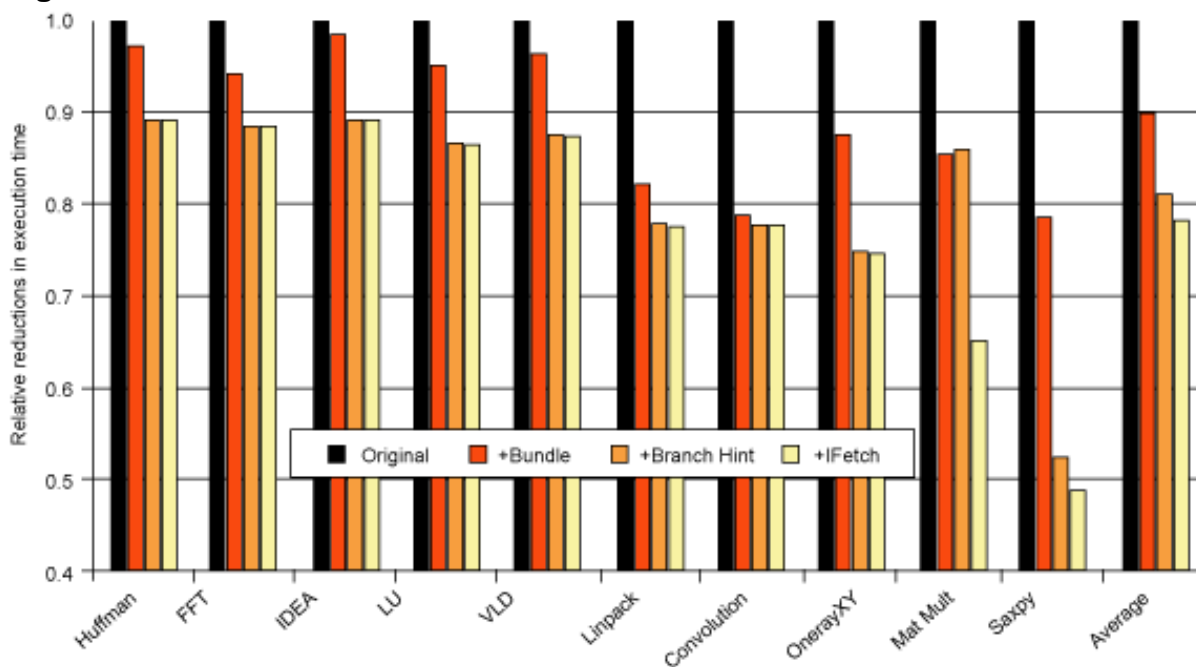
Scheduling and bundling as separate phases

In initial development, the scheduling and bundling phases were separate. The scheduler would find the best schedule using latencies, issue, and resource constraints, then the bundler would satisfy dual-issue and instruction-starvation constraints by adding NOPs. Unfortunately, this could result in poorly scheduled code, where the scheduler could have done a better job had it known about the need for an IFETCH operation.

In the current compiler, the scheduler and the bundler are tightly integrated on a cycle-per-cycle basis, allowing feedback from the bundler to guide the scheduler in making better decisions.

Effectiveness

Figure 7.



This figure shows the effect on the performance of a number of algorithms of the

bundling code, branch hinting, and IFETCH optimizations. The net result is about a 22% increase in performance on the average, although some algorithms are affected much more than others.

Section 7. Conclusions

The requirements of the SPE

The SPE has a number of unusual architectural concerns. It is a SIMD-only processor with no hardware branch prediction, instruction issue restrictions, and single-ported memory. This has imposed a number of duties on the compiler. The nature of the tasks makes them very hard for programmers to hand-tune.

Compiler involvement in high performance

The compiler is heavily involved in alleviating the unusual requirements of the SPE. The compiler handles scalar code automatically, both by hiding the complexity of real scalar code and by autovectorizing some code. The use of bundling to deal with the dual-issue restrictions can improve performance dramatically. A combination of using predication to handle simple if-then-else structures, and using good branch hinting, dramatically improves the performance of algorithms dependant on branching. Finally, management of the shared memory port between memory and instruction fetching helps reduce instruction starvation, improving the performance of some algorithms noticeably.

Overall effects

As of this writing, net improvement in performance is quite good across the board, with algorithms showing an average improvement of just over 20%. Some algorithms, such as matrix multiply and saxpy, show dramatic improvements, of 30%-50%. Continued development will likely yield further improvements.

Tune in next time

The next tutorial in this series will look in detail at the issues faced in extracting parallelism from code to run it efficiently on SIMD processors.

Acknowledgement

This tutorial series is based on the original presentation [Optimizing Compiler for the Cell Processor](#) given at PACT 2005 by Alexandre Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter Oden, Daniel Prener, Janice Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind of IBM Research.

This Part 2 is based on the section "SPE Optimizations."

Cell Broadband Engine is a trademark of Sony Computer Entertainment Inc.

Downloads

- Product: [Full-System Simulator for the CBE](#)
- Product: [XL C Alpha Edition for the CBE](#)
- Product: [CBE Software Sample and Library Source Code](#)
- Product: [GCC Toolchain for the CBE](#)
- Product: [CBE SPE Management Library](#)
- Product: [Linux kernel patch for the CBE](#)
- Product: [Fedora Core 4](#)
- Product: [SDK installation script](#)

Resources

Learn

- This [introduction to compiling for the Cell Broadband Engine tutorial series](#) is based on a presentation by [IBM Research](#) originally given at [PACT 2005](#).
- The IBM Research [Octopiler](#) is neither your grandfather's compiler, nor your grandfather's Oldsmobile.
- The [Cell Broadband Engine documentation](#) section of the IBM Semiconductor Solutions Technical Library lists specifications, user manuals, and articles of general interest -- including the [SPU Instruction Set Architecture documentation](#).
- Learn more about [OpenMP](#): a portable, scalable, cross-platform model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications.
- The IBM developerWorks [Cell Broadband Engine resource center](#) is your destination for all things Cell BE.
- Peruse the [developerWorks Power Architecture technology zone archives](#).

Get products and technologies

- Get [Cell-related downloads](#) including the IBM Full System Simulator, an evaluation copy of the Visual Age XL C compiler for Cell, and the Cell SDK from IBM alphaWorks.
- Download a copy of the [GCC compiler for Cell](#) from the Barcelona Supercomputer Center.
- Get Custom: [IBM Engineering & Technology Services](#) can help you with Cell- and custom-processor based solutions.
- Find more Power Architecture-related [downloads](#) in one page.
- Get a free subscription to the [Power Architecture Community Newsletter](#).

Discuss

- [Participate in the discussion forum for this content](#).
- Send a [letter to the editor](#).

About the author

Power Architecture editors

The developerWorks Power Architecture editors welcome your comments on this article. E-mail them at dwpower@us.ibm.com.

Trademarks

Cell Broadband Engine is a trademark of Sony Computer Entertainment Inc.