

# An introduction to compiling for the Cell Broadband Engine architecture, Part 5: Managing memory

## Analyzing calling frequencies for maximum SPE partitioning optimization

Skill Level: Intermediate

[Power Architecture editors](#)  
developerWorks  
IBM

07 Feb 2006

Fifth and last in the "[An introduction to compiling for the Cell Broadband Engine™ architecture](#)" series, this tutorial discusses techniques for managing data in the local store of the Synergistic Processor Elements (SPEs) of a Cell Broadband Engine (Cell BE) processor. Learn particular techniques such as double-buffering and maintaining a reasonably efficient software cache.

## Section 1. Before you start

### About this tutorial

This series provides an understanding of the Cell BE architecture, a basic intuition for programming issues on it, insight into the compiler challenges presented by it, and an understanding of the techniques and solutions proposed by the IBM compiler.

### Prerequisites

See the previous parts in this series:

- [Part 1: Overview](#): The Cell BE architecture and some of the issues faced in compiler design
  - [Part 2: Optimizing for the SPE](#): Optimizations used on the SPEs, such as how the compiler translates scalar code for a vector-only processor
  - [Part 3: Making the most of SIMD](#): How a compiler can effectively generate SIMD code for two different architectures (the SPE and VMX), accommodating the various technical constraints of the processors
  - [Part 4: Partitioning large tasks](#): How the compiler, or the user, can divide tasks up between the SPEs and the main processor
- 

## Section 2. Automatic memory management on the SPE

### Overview

This tutorial discusses the issues involved in memory management on the SPE units of the Cell BE processor. The primary focus is on the mechanisms used in the IBM compiler to automatically manage memory, with consideration given to the challenges faced in trying to maximize runtime performance of the whole Cell BE processor.

### Fitting sequential programs into local store, manually

You can manually develop a sequence of programs designed to fit into the local store on an SPE, each program fitting, along with its data, in 256KB of space. Once this is done, you would need a driver program to start an SPE thread and run your code. The set of programs would cooperate on the application, explicitly moving data in and out of local store buffers.

In this model, the programmer has complete control over the partitioning of functionality and the orchestration of data movement to achieve desired performance. The programmer uses the SPE compiler, the PPE compiler, and appropriate toolchain elements and embedding scripts to achieve this.

The downside of this approach is a huge imposition on the programmer, who has to keep track of all of these details. An automatic approach might be preferable.

## Fitting sequential programs into local store, using the compiler

Another option is a compiler-managed approach. The developer simply writes the program without regard to code or data size, focusing instead on algorithm design and correctness. This is then compiled using the SPE compiler with appropriate options.

An automizing compiler then does the partitioning of code and data. Code partitioning is done using overlays based on a call graph; data partitioning involves using a software cache, prefetching, and tiling. Only SPE toolchain elements are needed to do this, and the final program is executable from the PPE command line.

---

## Section 3. Code partitioning: Automatic overlay management

### Overview

Partitioning code and doing overlay management involves several key steps. First, a fixed-size buffer pool must be allocated in local store to hold instructions. The user program is then broken into pieces, each of which can fit in a single buffer, and the user startup is replaced with an overlay manager. Then, partitions are brought in as needed.

### Call-graph code partitioning

Call graph-based code partitioning uses the function as the primary partitioning unit. In general, a single function is small enough to fit into the local code buffer. Inter-partition transfers will happen on some calls and returns.

#### **Making a call graph**

A call graph is a directed graph in which nodes represent functions, and vertices represent function calls. Each node is assigned a weight reflecting its size (in instructions, typically), and each vertex is assigned a weight reflecting the frequency with which a given function calls another function. These graphs are developed during the interprocedural analysis phase of compilation.

## Function size

Call-graph partitioning faces challenges with very large single functions. Such functions are generally the result of aggressive optimizations, particularly function inlining and loop unrolling. Excessive unrolling and inlining should be avoided; they increase the number of code partitions, and thus the number of partition transitions. Currently, the user must control inlining and unrolling by the use of options.

When to inline depends somewhat on the size of the function being inlined. If the function is shorter than the function call sequence, inlining actually saves space! The default calling sequence on the SPEs is large enough that many simple functions should always be inlined.

A single function written by the user may also be very large. In this case, outlining the function allows it to be split into several smaller functions. Currently, outlining targets are outer loops, but any single-exit single-entry (SESE) code fragment could be outlined.

## The call-graph partitioning algorithm

The call-graph partitioning algorithm happens in the Interprocedural Analysis (IPA) phase of the compiler, when the whole program is available. The partitioning algorithm has two major stages.

### Building a partition affinity graph

The first major stage is building a partition affinity graph based on the global call graph. Each global call graph node becomes a node (partition) in the affinity graph. Each global call graph edge becomes an edge (partition transfer) in the affinity graph. Each call graph edge is weighted by expected call frequency.

Expected call frequency is based on three things: estimates based on static program analysis, profiling, and programmer input.

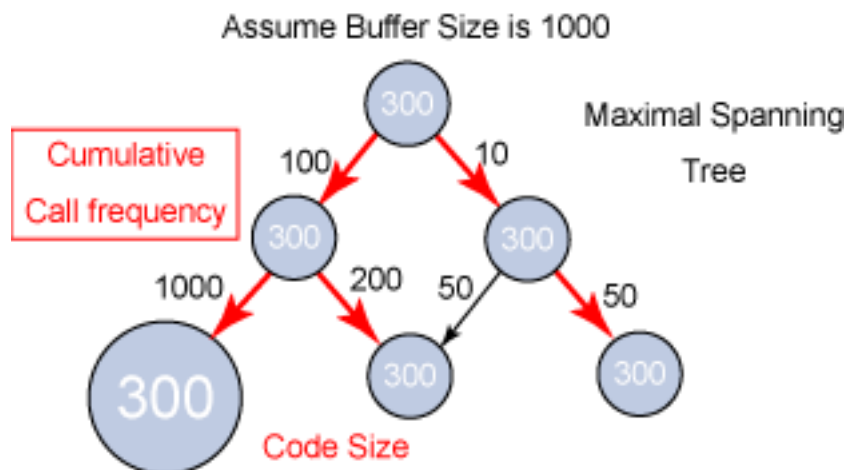
### Spanning trees

The next stage is applying a maximum spanning tree algorithm to the affinity graph. Edges are processed, starting at the leaves, in weighted order. Nodes are merged if their combined size is smaller than the buffer size.

At the end of this stage, nodes in the reduced graph are the program partitions.

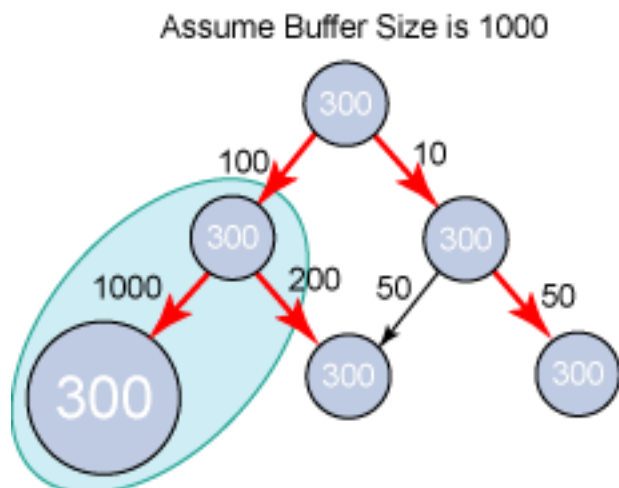
## An example of call graph partitioning.

To demonstrate how the call graph partitioning algorithm works, we'll use the following example graph. This demonstration assumes a buffer size of 1000 units, with code sizes measured in the same abstract units. All of the code partitions in the example are size 300, but the algorithm does not depend on this. Edges are weighted according to call frequency.



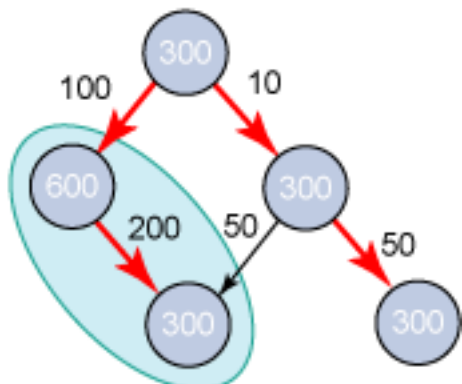
Thus, in this example graph, there are six nodes, each of size 300. The node in the lower left is called 1000 times by a single parent node, while the node in the lower middle is called 250 times; 200 by one parent, and 50 by another.

## The easy first combination



## Continuing combinations

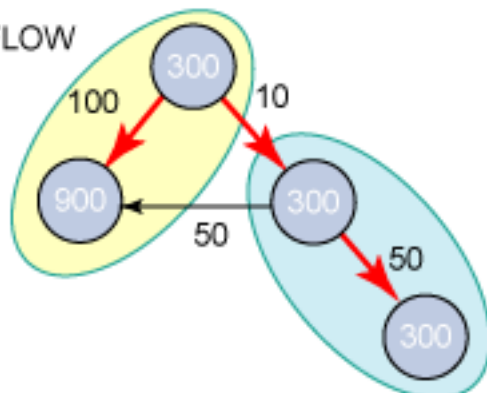
Assume Buffer Size is 1000



Once again, the highest-weighted edge can be combined. This leaves us with a graph in which one of the lines has changed directions:

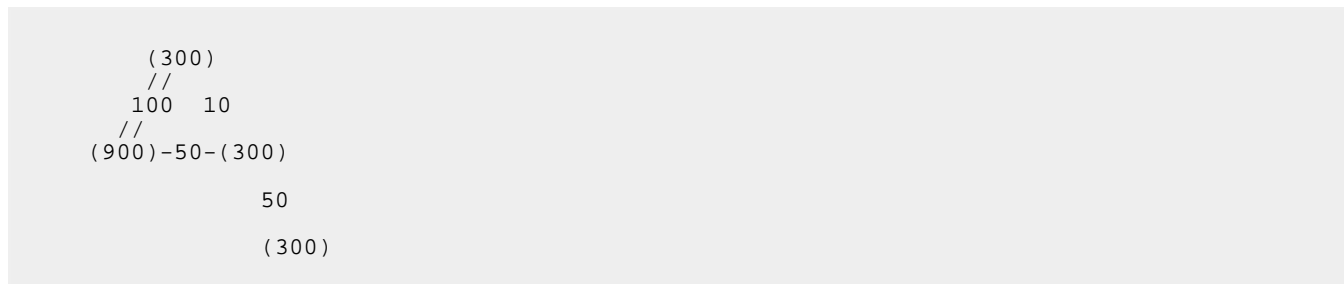
Assume Buffer Size is 1000

**BUFFER OVERFLOW**

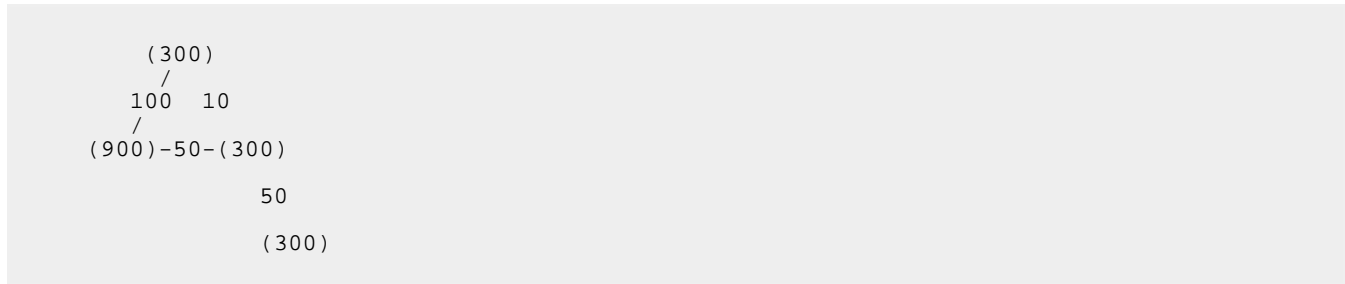


## Dealing with buffer overflow

The next obvious merge would result in a buffer overflow; it would result in putting 1200 items in a 1000-item buffer.



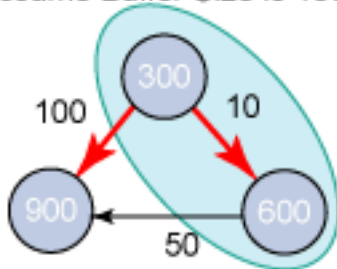
Instead, then, the next highest-weighted edge will be combined:



## Finishing up the algorithm

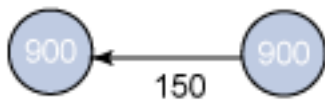
Now the two highest-weighted edges cannot be combined, because they would result in overflow. However, the other edge can be combined:

Assume Buffer Size is 1000



At the end of the algorithm, we have only two nodes, each with 900 units of storage, with 150 calls between them.

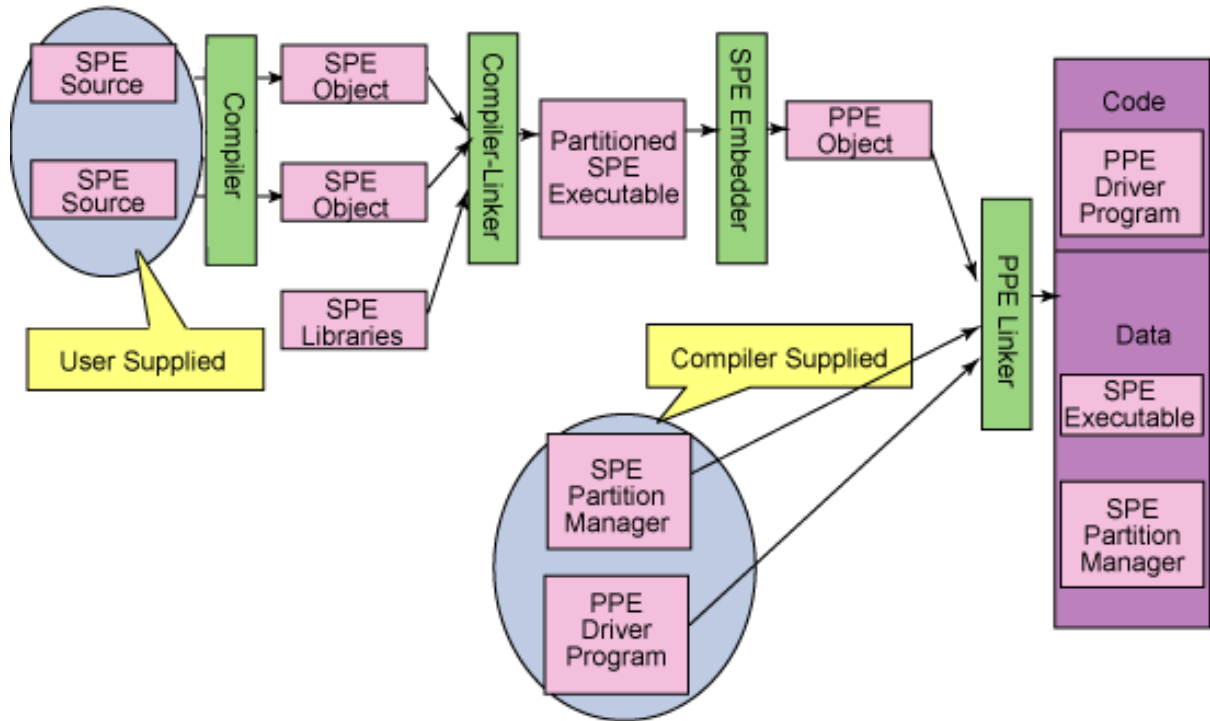
Assume Buffer Size is 1000



Given that 150 cross-buffer calls is fewer calls than were eliminated by each of the first two combinations, the algorithm can reasonably be called a success.

## Compiling and binding a Cell BE program manually

### Manual compilation and linkage



## How manual compilation works

Source specific to the SPEs is compiled into objects, linked together with libraries, and built into SPE executables by a linker. An additional phase then converts these executables into embeddable objects for use by the PPE linker. On the PPE side, source is once again compiled into objects, and linked against libraries; however, the linker also incorporates the embeddable objects created by the SPE embedder. The result is a normal executable, containing code and data, but some of the data is actually SPE executables.

## Manual execution of a stand-alone SPE program

An SPE program is not executed directly; rather, it is embedded into a PPE object file for a wrapper program. It is embedded in the data section of the PPE program, and there's a symbol for it in the PPE program's symbol table.

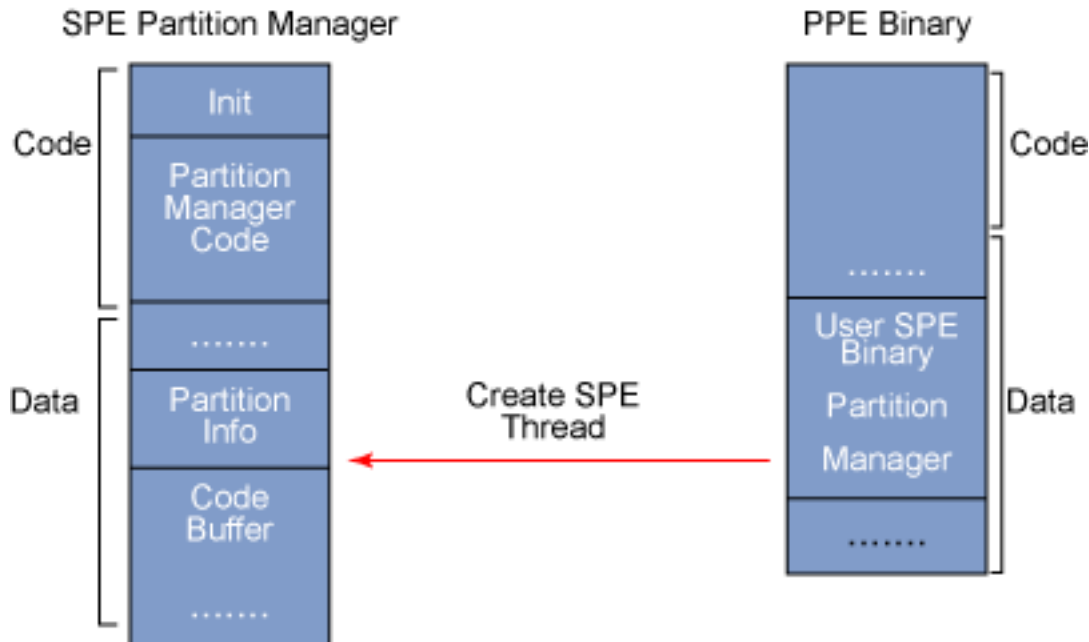
The PPE driver program is executed directly on the PPE. It locates the SPE executable image, creates an SPE thread with that image, and starts the SPE thread. In some cases, it may then cooperate with the SPE thread, providing system services and other functionality for it.

The PPE program might be anywhere from very simple to fairly elaborate; any PPE program can run an SPE program. On the other hand, if the PPE program is starting

multiple SPE threads, then the SPE program isn't really standalone anymore.

## Compiling and binding a partitioned SPE program

### Using the partition manager



The user-provided SPE source is compiled into SPE objects. When the SPE objects are being linked together, the TPO link phase partitions SPE code, dividing it into multiple w-code partitions. These are fed individually through the TOBEY scheduler to create SPE objects, which are linked by the SPE linker; the resulting SPE executable is partitioned.

The SPE embedder then creates a PPE object, which is linked together with a compiler-supplied SPE partition manager, and the compiler-supplied PPE driver code, to create a PPE driver program.

## Execution of a compiler-partitioned SPE program

The process of running a compiler-partitioned SPE program has some similarities to the process of running a manually partitioned program. The PPE driver program is comparatively complex, because it's a generic driver capable of handling a variety of SPE program styles.

## The function of the PPE driver program

The PPE driver program locates the embedded SPE partition data manager program

in its data section. The PPE driver then creates an SPE thread loading this image, starts the SPE thread, starts a service thread to cooperate with the SPE thread (providing system calls), and waits for the SPE thread to terminate.

## The function of the SPE partition manager

The SPE partition manager locates the SPE image for the user-supplied code, loads the initial partition, and branches to it. The user-supplied code is not loaded by the PPE driver; the SPE requests the code it needs, when it needs it.

## Possible extensions to the PPE driver / SPE manager model

Many extensions are possible. For instance, it would be possible to run additional user-supplied code in the PPE thread. The compiler could also provide support for functional partitioning.

## Compilation and linking of a user SPE program

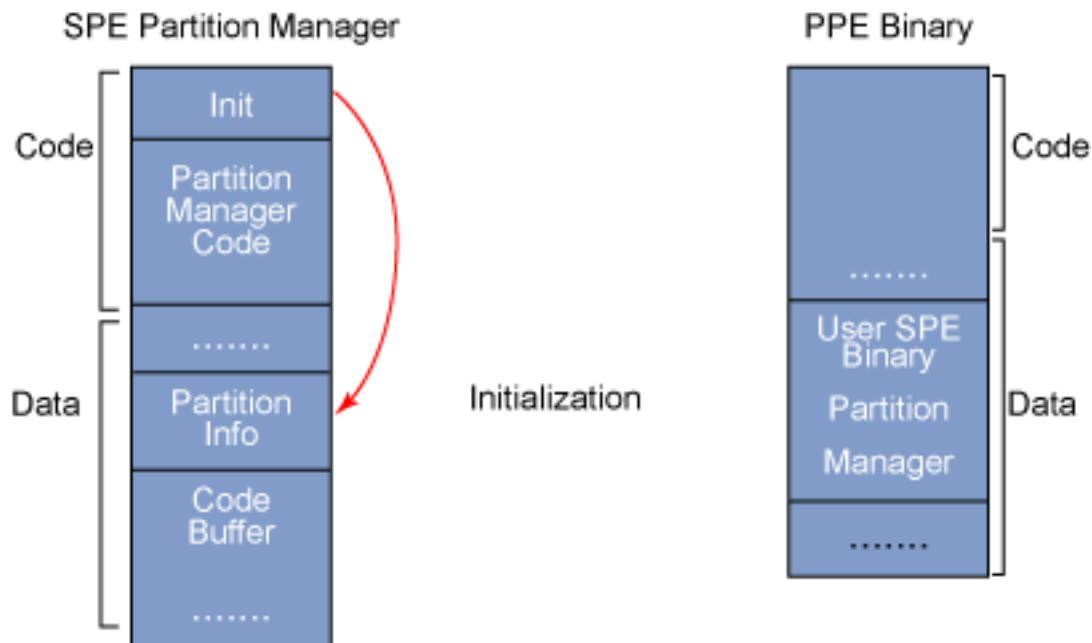
The user-provided code is linked against the SPE partition manager. When it refers to functions defined in the partition manager, they are resolved at an absolute address. In fact, the SPE partition manager doesn't need to be linked into the user SPE executable, but it will reside at a fixed address at runtime.

A new compilation unit is created for each program partition. This guarantees that functions in the same partition are contiguous. To fetch a partition, the SPE partition manager needs to know the offset and size of the partition; this information can be extracted from the program binary. While the partition manager is linked at a fixed address, all partitions are linked to be position-independent, so they can be loaded at any address.

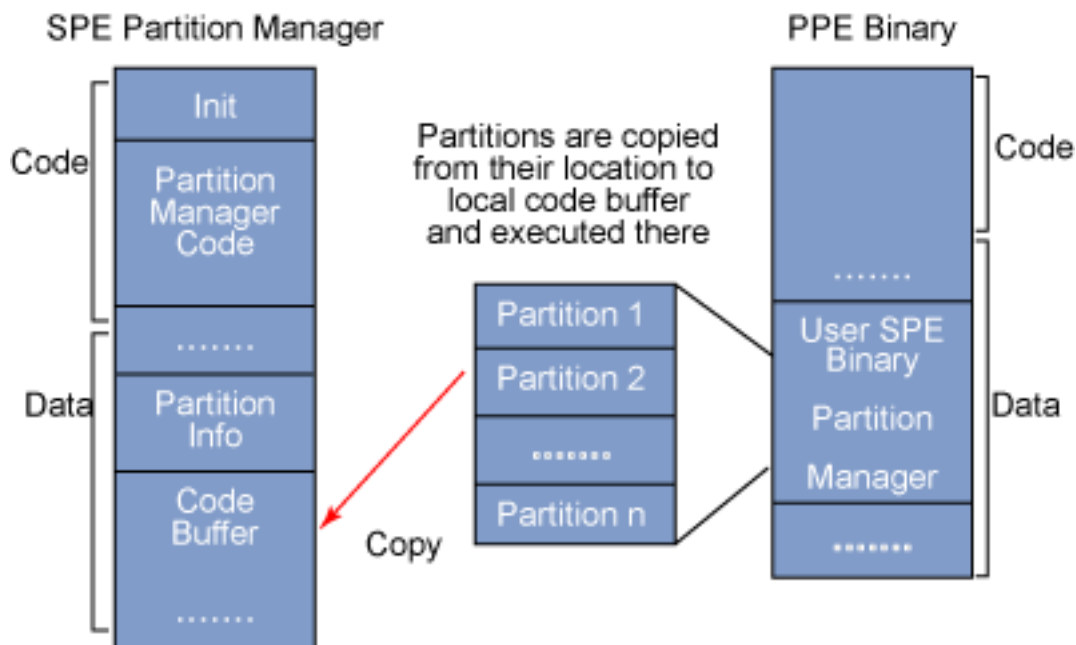
---

# Section 4. Execution of a partitioned SPE program

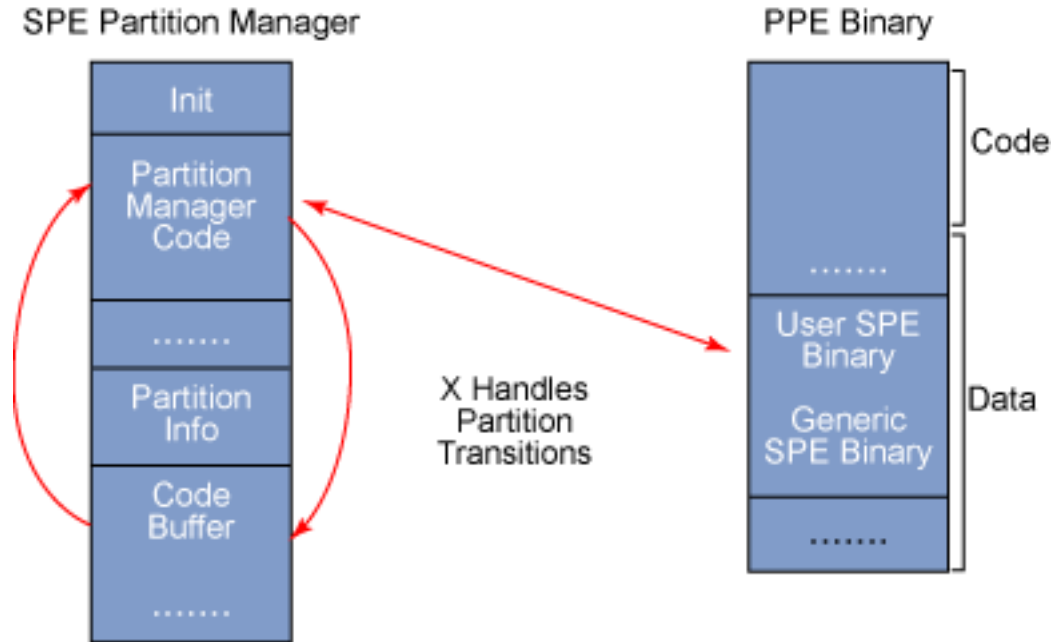
## Initializing a code partition



**Copying code into a code buffer**



**Handling partition transitions**



## Creating an SPE thread

The first thing the PPE binary does with an SPE is create a new SPE thread, loading the SPE partition manager program onto the SPE. The SPE partition manager has a small amount of initialization code, and also contains the various partition manager functions that partitioned code has been linked against. The rest of its local store is reserved for data, such as partition information, a code buffer, and any needed data buffers. The user SPE code remains in the data segment of the PPE program, for now.

## SPE program initialization

The SPE partition manager's initialization code sets up the partition information it needs to do its work. The PPE doesn't need to be involved in this process, and can go ahead and do whatever else it wants.

Once the partition information is configured, the partition manager can begin copying partitions over from their locations in the data segment of the PPE binary to local code buffers, in preparation for local execution.

## SPE program actively running

After some code has been loaded, the actual user code begins running. As needed, it makes calls to the partition manager code, which loads new partitions of the code

as needed, handling partition transitions automatically. The user's original code doesn't need to know about this; the compiler's partitioning and overlaying code takes care of it.

---

## Section 5. Partition manager implementation

### Necessary program transformations

The partition manager has to take over on inter-partition calls and returns, to make sure that the code being called is loaded. Inter-partition calls are transformed into calls to a stub function in the partition manager, taking one extra argument, the address of the target function. The partition manager then translates this appropriately, and loads the partition containing the target function if necessary.

### Partitioning third-party libraries

You'll face several issues in partitioning third-party libraries. In general, third-party libraries come without source code, so the code in them cannot be transformed, making inter-partition calls impossible.

To solve this, each library partition is self-contained. It cannot make calls to other library partitions and is compiled with relative addressing, so it can be moved around freely.

### Implementation of library partitioning

The implementation of partitioning code for the library involves a certain amount of extra work. First, the compiler duplicates the linker's symbol-resolution logic. The compiler can then scan library symbol tables to find external definitions and references, and figure out which references are resolved by which definitions. If one object references another, they can be put into the same partition.

### Performance optimizations

You can improve the performance of compiler-managed partitioning in a number of ways. One important step is profiling to get accurate call edge frequencies. The heuristics available in static analysis are not always accurate. This is especially true

for code that uses function pointers, where only profiling can reliably tell which calls are really made at runtime.

Two-pass partitioning can produce more accurate estimates of function code size. The code size of a function is normally estimated statically, and this estimate must be conservative. The compiler back-end often makes significant transformations, such as code scheduling, inserting NOPs (Not an Operation), unrolling loops, and inlining other functions. In a two-pass approach, each function is made a partition in the first pass, the code size is collected, and in the second pass, partitioning can be done with more accurate information about code size.

## Even more performance optimizations

A cheap but surprisingly effective optimization is to use two buffers for partitions. When transferring from one partition to another, if the second partition is read into a separate buffer, it's quite likely that the next transfer is back to the first buffer, which is already loaded.

Prefetching is a more challenging approach. There's a huge latency (hundreds of cycles) to look at, so a successful prefetching policy must be an aggressive one, and will probably engage in speculative prefetching. However, an overly aggressive policy can waste a great deal of bandwidth fetching code that's never run, which is also bad. It is hard to strike a good balance that improves results.

This has some conceptual overlap with the issues in deciding when to issue branch hints; see [Part 3: Making the most of SIMD](#).

---

## Section 6. Partitioning data

### Overview

The previous discussion focused on partitioning code. It is also possible, and necessary, to partition data. Many programs have data sets much larger than the available local store of an SPE. The simple solution is to use system memory, which is a plentiful resource. However, since the SPU doesn't have load/store access to system memory, data must be transferred back and forth. The compiler can automatically manage the transfer of data between system memory and local store. This is called data partitioning, by analogy to code partitioning.

## The single-source assumption

One simple approach to data access is to simply assume that all data lives in system memory, and use DMA to access data, transferring it to the SPE for a load, and from the SPE after a store. This has critical flaws. First, it imposes unacceptable latency (about 300 cycles per load or store), and second, MP would require locking on every reference. This is obviously unworkable.

## Accessing globals in system memory

This example looks at a possible implementation of a very simple program on an SPE using DMA to get memory. The sample program looks something like this:

```
float x, y, z;
int main() {
    x = y + z;
}
```

Implemented on an SPE, using a very naïve approach to system memory, this would be split up into several components. The PPE thread would simply start the SPU thread and wait, but the values `x`, `y`, and `z` would be stored in the memory allocated to it. The SPE code has room for copies of the values in question in local store, as well as records of their main memory addresses. We refer to the local store copies as `t.x`, `t.y`, and `t.z`, and the addresses as `.x`, `.y`, and `.z`.

## Setting up access to global variables

The SPE embedder builds code to make a table of the addresses of the real memory objects that need to be accessed whenever these variables are touched. The code that starts the SPE thread fills these addresses in before starting the SPE code. Thus, the values `.x`, `.y`, and `.z` are filled in from the relocation table entries for the main-memory variables `x`, `y`, and `z`.

## What the SPE actually does

Once the values are filled in, the SPE's code is roughly structured like this:

```
load r100 = .y
call dma_get t.y = r100
load r101 = t.y
load r102 = .z
call dma_get t.z = r102
```

```
load r103 = t.z
add r104 = r101,r103
store t.x = r104
load r105 = .x
call dma_put =t.x, r105
```

As you can see, this code is painfully inefficient.

## Reducing the overhead of accesses to system memory

Rather than executing a DMA transaction for every variable access, we would like to bring data over in larger pieces, and keep it in local store for some portion of its lifetime. Several techniques can accomplish this.

The first technique is to prefetch predictable references and accumulate writes; both of these are direct attempts to reduce the number of transactions and reduce the effective latency of remaining transactions.

Another useful technique is a software cache. This is managed on demand, but leverages spatial and temporal locality. It does require additional instructions inline, so it is essentially a fall-back strategy.

Both of these techniques can have improved reuse using tiling.

## A closer look at prefetching

The following code samples illustrate how prefetching works. We begin with a simple loop:

```
for (i = 0; i < 100000; ++i)
    a[i] = b[i] + c[i];
```

This code will now be expanded to use prefetching to reduce latency. Although this code looks simple, if the data it uses were stored in main memory, each access in the above would look like the previous example, so each pass through the loop would involve two DMA reads and one DMA write.

## Blocking prefetching

The first, simple, version of prefetching doesn't really eliminate blocking, but makes it less frequent. (Note: The `dma_get` function is counting in bytes, which is why it takes a 400-unit `dma_get` or `put` to move 100 32-bit objects.)

```

for(i = 0; i < 100000; i += 100) {
    dma_get(b', b[i], 400);
    dma_get(c', c[i], 400);
    for(ii = 0; ii < 100; ++ii)
        a'[ii] = b'[ii] + c'[ii];
    dma_put(a[i], a', 400);
}

```

This function still needs to wait for DMA, but it needs DMA operations much less often.

## Software-pipelined prefetch

The next version shows a more carefully implemented (and somewhat more complicated) software-pipelined prefetch.

```

dma_get(b', b[0], 400);
dma_get(c', c[0], 400);
for(i = 0; i < 99900; i += 100) {
    dma_get(b'', b[i+100], 400);
    dma_get(c'', c[i+100], 400);
    for(ii = 0; ii < 100; ++ii)
        a'[ii] = b'[ii] + c'[ii];
    dma_put(a[i], a', 400);
    swap(a', a'');
    swap(b', b'');
    swap(c', c'');
}
for(ii = 0; ii < 100; ++ii)
    a'[ii] = b'[ii] + c'[ii];
dma_put(a[i], a', 400);

```

In this program, past the initial operation, each dma operation is queued up as long as possible before it's needed. Thus, the b'' array is used to prefetch, not the data that will be operated on in this pass through the outer loop, but the data that will be needed to start the next pass; this gives a very good chance that the DMA operations will be complete before the data are needed.

In the case where operations can be predicted, it's fairly easy to prefetch. However, in some cases data access patterns are irregular:

```

for (i = 0; i < 100000; ++i)
    a[i] = b[i] + c[i] * d[f(i)];

```

In this case, b and c can be prefetched, but the access pattern in d cannot be predicted. At first, we seem to be thrown back on the naive implementation;  $d[f(i)]$  must be fetched on each iteration, with a consequent huge slow-down of

the loop. Every access to `d` requires a high-latency access to main memory.

## Software caching

While the SPEs have no hardware cache, it is possible to implement a small software cache by explicitly referring to it. For instance, the above example can be implemented as follows:

```
for (i = 0; i < 100000; ++i) {
    t = cache_lookup(d[f(i)]);
    a[i] = b[i] + c[i] * t;
}
```

A sample implementation of `cache_lookup` might look like this:

```
inline vector cache_lookup(addr) {
    /* fetch the value if we haven't got it already */
    if (cache_directory[addr & key_mask] != (addr & tag_mask))
        miss_handler(addr)
    /* return the value */
    return cache_data[addr & key_mask][addr & offset_mask];
}
```

In this implementation, `miss_handler` will get the required data, and some suitable quantity of surrounding data, through DMA. Higher degrees of associativity can be supported, possibly for little extra cost on an SIMD processor.

## Combining prefetch with software cache

You can combine these techniques. This code sample uses both predictable and irregular data accesses:

```
for (i = 0; i < 100000; ++i)
    a[i] = b[i] + c[i] * b[f(i)];
```

This can be expanded into a function using both prefetch and software caching.

```
for(i = 0; i < 100000; i += 100) {
    dma_get(b', b[i], 400);
    dma_get(c, c[i], 400);
    for(ii = 0; ii < 100; ++ii) {
        t = cache_lookup(b[f(i)]);
        a'[ii] = b'[ii] + c'[ii] * t;
    }
}
```

```
    dma_put(a[i], a', 400);  
}
```

Prefetching will possibly already have obtained  $b[f(i)]$ , so it must update the cache directory, and miss handling must not evict prefetched data. In this particular case, evicting prefetched data would only cause cache misses, but if the last term were  $a[f(i)]$ , the miss handler could theoretically throw away the results we're trying to calculate.

---

## Section 7. Software caching in more detail

### Overview

A reasonably well-implemented software cache can dramatically improve performance of many algorithms.

The software cache described here is made up of two arrays. The first, the tag array, contains the comparands for the address lookups, pointers to the data lines, and "dirty bits" used for MP support. The data array contains the data lines.

This cache design is four-way set associative, with 128 lines of 128 bytes in each set, although these parameters can be changed.

The total size used for the cache (not including code) is 16K for tags and 64K for data, for a total of 80K of storage.

### Software cache lookup

The cache lookup code is inserted inline at each read or write reference to a cacheable variable. Because the code is inserted inline, it is optimized along with all the other instructions. The branch to the miss handler is treated as a regular instruction throughout optimization and expanded only at the very end of compilation. The miss handler uses a tailored register convention so that it has no impact on the hit path.

### How the software cache lookup works

The software cache lookup procedure takes a bit of explanation. To start with,

presume an SIMD register containing a desired data address in the preferred slot (slot 0 of the register). The desired address is a 32-bit value. The first 18 bits are used as a comparand, followed by a seven-bit index (into the 128 lines of the cache) and a seven-bit offset (the offset within a given cache line).

## Obtaining byte offsets and comparands

To perform a cache lookup, you need three things. You need two SIMD registers, one containing four copies of the 18-bit comparand, and one containing four copies of the seven-bit byte offset into the cache line. You also need the byte index of the tag array entry. The SIMD registers can be obtained by using a splat instruction to populate an SIMD register with four copies of the data address, then making two copies, one bitwise-anded with a splat of 0x7F, and one bitwise-anded with a splat of ~0x7F. The byte index into the tag array is obtained by bitwise and with 0x3F80. (0x3F80 is the seven bits above 0x7F.)

## Looking up the right entry in the tag array

Given the address of the tag array, adding the byte index into the tag array (data address & 0x3F80) to it yields the address of the tag array entry that might hold the desired cache line. In fact, the tag array entry holds tags, and data line addresses, for four sets. The tag array entry is loaded into two SIMD registers; the first contains four comparands matching the tag array entry we're looking at; the second contains the corresponding data line addresses.

## Comparing with the tag array

Now, you compare the tag array comparands to your SIMD register of comparands from the data address. If there is a match, the data in question is in the cache; you now rotate the register containing the data line addresses so the corresponding address is in the preferred slot, and add that register to your SIMD register of byte offsets. The preferred slot now holds the address of the cached data, ready to be loaded in and returned.

## MP support in the software cache

When a sequential program is running on a single SPE, you don't keep track of what is written in the lines. The miss handler always writes the whole line back on eviction.

For MP, though, several SPEs might write parts of a line. The solution is to keep track of what is written by using "dirty bits" -- one bit for each byte in the line, kept in

the tag array. The miss handler only transfers dirty bytes. The memory flow controller (MFC) handles merging of writes.

## Miss handler

The miss handler is called when a requested line is not in the cache, and handles it by loading the missing data. First, it chooses one of the four lines in the set to evict; currently, this choice is made at random. The dirty bytes of the line are written back to system memory. In the uniprocessor case, the whole line is written. The DMA locking instructions are used to lock the line in system memory; the whole line is locked, rather than separate words within it. The newly requested line is transferred from system memory, the tag array is updated, and control returns to the inline lookup code in the requestor.

## Conclusions

The software cache mechanism allows the comparatively efficient use of variables stored in main memory and shared between threads. Coupled with other techniques to handle other workloads, such as efficient use of double buffering to stream data in and out while processing continues, the Cell BE compiler offers a reasonable way for developers to maintain good productivity while getting reasonable efficiency from the SPE units.

---

## Section 8. That's all, folks!

### Congratulations!

This tutorial ends this series looking at the compiler technology used to target the Cell Broadband Engine processor. For more ideas of things to read up on, see the [Resources](#) in this tutorial (and in the others).

### 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, Byoungso So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael

Gschwind of IBM Research.

This Part 5 is based on the section "Automatic Memory Management on the SPE."

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](mailto:dwpower@us.ibm.com).