SPU MFC-DMA Programming

Mark Nutter
mnutter@us.ibm.com

1 Introduction

In this paper we provide an overview of SPU MFC-DMA programming. The MFC is basically a data transfer engine, intended principally for the movement of data between main and local storage.

Much of this material can be found in the *Broadband Processor Architecture, Books I-III*. This paper includes new material targeted specifically to assembly and C language programmers who are interested in initiating MFC-DMA transfers from a program executing on an SPU.

The following section contains background material on the Broadband Processor Architecture. In section 3 we describe the MFC-DMA command types. In section 4 we describe the steps required to initiate an MFC-DMA transfer from an SPU. In section 5 we describe various conditions that affect MFC-DMA transfers and SPU program flow. In section 6 we describe MFC-DMA list operations. In section 7 we outline basic procedures for overlapping data transfers with computation on an SPU.

2 BPA Overview

Within the Broadband Processor Architecture (**BPA**), a processing unit (**PU**) core is a general purpose CPU with access to system management resources.

A synergistic processing unit (**SPU**) core is a somewhat less complex computational unit, which does not perform any system management functions. The SPU has a highly optimized single instruction, multiple data (**SIMD**) instruction set, and is intended to enable applications with a high computational unit density.

Two types of storage are defined by the BPA, and these are referred to as *local storage* and *main storage*. Local storage (**LS**) consists of a separate memory region that is associated with an SPU. The SPU can only execute instructions, including data load/store operations, from within its own associated LS. An SPU program references its own LS using a local address (**LA**). Main storage may be shared by all processors and I/O devices in a system. Each LS is also assigned a real address (**RA**) within the system's memory map. This allows privileged software to map LS areas into the effective address (**EA**) space of a process. EA's are translated and protected using the PowerPC storage model.

The memory flow controller (**MFC**) serves as a data transfer engine for an SPU. MFC direct memory access (**DMA**) commands can be used to initiate a transfer from main storage into a local storage area, and visa versa. The MFC supports transfer sizes of 1, 2, 4, 8[1], and multiples of 16-bytes, with a maximum transfer size of 16 kb. Peak performance is achieved for transfers where both the EA and LA are 128-byte aligned, and the size is an even multiple of 128-bytes.

Each MFC maintains and processes two queues of commands, one for PU-initiated transfers and another for SPU-initiated transfers. Within the Broadband Engine (**BE**), the PU queue depth is eight elements, while the SPU queue depth is sixteen elements.

The SPU interacts with the MFC through the SPU channel interface (**CI**). The SPU utilizes channel numbers 0-31 to interface with the MFC, manage external events, query or set the decrementer, determine signal notification status, and interact with an SPU's mailbox resources. SPU instructions allow a program to write to a designated channel (**wrch**), read from a channel (**rdch**), or read a channel counter (**chcnt**).

3 DMA Command Types

This section describes the DMA command types, which consist of four major categories: *PUT*, *GET*, *Storage Control*, and *Synchronization*.

3.1 PUT Commands

The following table lists the DMA PUT commands, which move data from LS to EA.

Figure: DMA PUT Commands

Command	Description
put	move data from LS to EA
puts	move data and start SPU after xfer (**PU only**)
putr	move data w/ L2 scarf hint
putf	move data w/ fence
putb	move data w/ barrier
putfs	move data w/ fence & start SPU after xfer (**PU only**)
putbs	move data w/ barrier & start SPU after xfer (**PU only**)
putrf	move data w/ fence & L2 hint
putrb	move data w/ barrier & L2 hint
putl	move data using DMA List (**SPU only**)
putlf	move data w/ fence using DMA List (**SPU only**)
putlb	move data w/ barrier using DMA List (**SPU only**)
putrlf	move data w/ fence & L2 hint using DMA List (**SPU only**)
putrlb	move data w/ barrier & L2 hint using DMA List (**SPU only**)

3.2 GET Commands

The following table lists the DMA GET commands, which move data from EA to LS.

1. Transfer sizes of 1, 2, 4, and 8-bytes will utilize a local store offset within the LA quadword (16-bytes) defined by the least significant 4-bits of the EA.

Figure: DMA GET Commands

Command	Description
get	move data from EA to LS
gets	move data & start SPU after xfer (**PU only**)
getf	move data w/ fence
getb	move data w/ barrier
getfs	move data w/ fence & start SPU after xfer (**PU only**)
getbs	move data w/ barrier & start SPU after xfer (**PU only**)
getl	move data using DMA List (**SPU only**)
getlf	move data w/ fence using DMA List (**SPU only**)
getlb	move data w/ barrier using DMA List (**SPU only**)

3.3 Storage Control Commands

The following table lists the DMA Storage Control commands.

Figure: DMA Storage Control Commands

Command	Description
sdcrt	bring a range of EAs into SL1 (perf. hint for GET)
sdcrtst	bring a range of EAs into SL1 (perf. hint for PUT)
sdcrz	zeros the contents of a range of EAs
sdcrst	stores the modified contents of a range of EAs
sdcrf	stores the modified contents of a ranges of EAs and invalidate block

3.4 Synchronization Commands

The following table lists the DMA Synchronization commands[1].

Figure: DMA Synchronization Commands

Command	Description
getllar	get lock line and create reservation (**SPU only**)
putllc	put lock line condition on reservation (**SPU only**)
putlluc	put lock line unconditional (**SPU only**)
sndsig	update signal notification of I/O device or another SPU
sndsigf	update signal notification w/ fence
sndsigb	update signal notification w/ barrier
barrier	order all prior DMA commands w/ respect to subsequent commands in the queue.
eieio	order the storage effects of DMA commands w/ respect to other processors and mechanisms for I/O
sync	order the storage effects of DMA commands w/ respect to other processors and mechanisms

4 DMA Command Issue

In this section we describe the steps required to initiate an DMA transfer from a program executing on the SPU.

To enqueue an DMA command from the SPU, the command parameters must first be written to the DMA *command parameter channels* (16-20). The DMA command parameters are retained in these channels until a write to the DMA *enqueue command channel* (21).

1. Full treatment of DMA synchronization commands is beyond the scope of this paper.

Figure: Initiate DMA (SPU-assembly)

```
# Inputs:
#   $0 contains LA
#   $1 contains EA-high 32-bits
#   $2 contains EA-low 32-bits
#   $3 contains transfer size in bytes
#   $4 contains tag id between [0..31]
#   $5 contains DMA command

# Transfer using 64-bit EA
    wrch $mfc_ls_addr[1], $0
    wrch $mfc_ea_high[2], $1
    wrch $mfc_ea_low, $2
    wrch $mfc_dma_size, $3
    wrch $mfc_tag_id, $4
    wrch $mfc_cmd_queue, $5
```

1. The *$mfc* channel names are outlined in the SPU Assembly Language Specification.
2. This parameter is optional, and will be set to zero if not written.

Figure: Initiate DMA (SPU-C)

```
#include <bpa_mfc.h>

void *la;
unsigned int size;
unsigned int tag;
union {
    unsigned long long ull;
    unsigned int ui[2];
} ea;

/* Transfer using 32-bit EA */
spu_mfcdma32(la, ea.ui[1], size, tag, MFC_GET_CMD);[1]

/* Transfer using 64-bit EA */
spu_mfcdma64(la, ea.ui[0], ea.ui[1],
             size, tag, MFC_PUTF_CMD);
```

1. The *spu_mfcdma* intrinsics are described in the SPU C/C++ Language Extensions.

5 DMA Conditions

In this section we describe various conditions that affect DMA transfers and SPU program flow. These include determining the completion status of DMA tag groups, as well as determining the number of entries available in the DMA command queue.

5.1 DMA Tag Group Completion

Each DMA command is tagged with a 5-bit identifier, supporting values in the range of 0-31. The same tag identifier may be used for multiple DMA commands. A set of commands with the same identifier is defined as a *tag group*. Software can use this identifier to check or wait on the completion status of all queued commands in one or more tag groups.

Two basic procedures are supported with respect to determining the completion status of one or more commands using tag groups:

- Poll tag group update status
- Wait for tag group update or wait for external event

5.1.1 Poll Tag Group Update Status

The basic procedure for polling the completion of an DMA command or group of DMA commands is as follows:

1. Set the *tag group mask* (channel 22).
2. Request immediate *tag status update* (write 0x0 to channel 23).
3. Read *tag status* (channel 24).
4. Repeat steps 2 & 3 as necessary.

Figure: Poll Tag Status (SPU-assembly)

```
# Inputs:
#   $0 contains tag mask
# Outputs:
#   $1 contains completed tag status

# Set tag group mask
    wrch $mfc_wr_tag_mask[1], $0

# Set up for immediate tag status update.
    il $1, 0

# Repeat as necessary.
repeat:
    wrch $mfc_wr_tag_update, $1
    rdch $1, $mfc_rd_tag_status
    brz $1, repeat
```

1. Once written, the tag mask is retained until changed by another write to the tag mask.

Figure: Poll Tag Status (SPU-C)

```
#include <bpa_mfc.h>

/* Declare tag_id. */
unsigned int tag_id = 0;

/* Construct tag_mask from tag_id. */
unsigned int tag_mask = 1 << tag_id;

/* Set tag group mask */
spu_writech(MFC_WR_TAG_MASK, tag_mask);

/* Repeat as necessary. */
do {
} while (!spu_mfcstat(MFC_TAG_UPDATE_IMMED));
```

5.1.2 Wait for Tag Group Update

The basic procedure for waiting on a conditional tag event (one or more tag group completions) is as follows:

1. Set the tag group mask.
2. Request conditional tag status update. If requesting completion status for *any* tag in the mask, use 0x1; if requesting completion status for *all* tags in the mask, use 0x2.
3. If only waiting for conditional update, read tag status.
4. If waiting for any one of multiple events, unmask *tag status update* in the *external event mask* (channel 1) and read from the *external event status* (channel 0)[1].

Figure: Wait for Tag Status (SPU-assembly)

```
# Inputs:
#   $0 contains tag mask
# Outputs:
#   $1 contains completed tag status

# Set tag group mask
    wrch $mfc_wr_tag_mask, $0

# Set up for ANY tag status update.
    il $1, 0x1

# Wait for conditional tag status update.
    wrch $mfc_wr_tag_update, $1
    rdch $1, $mfc_rd_tag_status
```

1. Full treatment of SPU external events is beyond the scope of this paper.

Figure: Wait for Tag Status (SPU-C)

```
#include <bpa_mfc.h>

/* Use tag_mask from previous example */
spu_writech(MFC_WR_TAG_MASK, tag_mask);

/* Wait for ALL ids in tag group to complete */
spu_mfcstat(MFC_TAG_UPDATE_ALL);
```

5.2 DMA Queue Space

On the BE implementation of BPA, each SPU has a 16 element DMA command queue. As the SPU submits DMA commands to the MFC, this queue may become full. Before subsequent commands can be issued, space must become available in the DMA command queue.

Two basic procedures are supported with respect to waiting for available space in the DMA command queue:

- Poll for queue space
- Wait for queue space or wait for external event

5.2.1 Poll for Queue Space

The basic procedure for polling for available space in the DMA command queue is as follows:

1. Read the channel count for the DMA command enqueue (channel 21).
2. Repeat until the desired amount of queue space is available.

Figure: Poll Queue Space (SPU-assembly)

```
# Outputs:
#   $0 contains available DMA queue space.

# Repeat as necessary.
repeat:
   chcnt $0, $mfc_cmd_queue
   brz $0, repeat
```

Figure: Poll Queue Space (SPU-C)

```
#include <bpa_mfc.h>

do {
} while (!spu_readchcnt(MFC_CMD_QUEUE));
```

5.2.2 Wait for Queue Space

The basic procedure for waiting for available space in the DMA command queue is as follows:

1. If only waiting for space in the DMA command queue, simply enqueue the DMA command as usual. The SPU will block on the channel write to the DMA command enqueue (channel 21) until space becomes available.
2. If waiting for any one of multiple events, unmask *DMA queue vacancy event* in the external event mask (channel 1), and read from the external event status (channel 0).

Because of the blocking semantics of the DMA command enqueue (channel 21), the assembly and C language code sequences are the same as for regular DMA command issue.

6 DMA Lists

This section describes DMA *lists*, which allow programs to construct a list of transfer size and EA pairs that are stored sequentially in LS, and are used to control DMA transfers. DMA lists basically implement scatter-gather function between EA to LS. DMA list commands may only be initiated by the SPU.

Figure: DMA List Commands

Command	Description
putl	put data using DMA List
putlf	put data w/ fence using DMA List
putlb	put data w/ barrier using DMA List
putrlf	put data w/ fence & L2 hint using DMA List
putrlb	put data w/ barrier & L2 hint using DMA List
getl	get data using DMA List
getlf	get data w/ fence using DMA List
getlb	get data w/ barrier using DMA List

The first word in each DMA *list transfer element* contains the transfer size and an optional *stall-and-notify* flag. The second word contains the low order 32-bits of the EA.

Figure: DMA List Element (SPU-C)

```
#define SPU_DMA_LIST_STALL_FLAG (1<<31)
struct spu_dma_list_elem {
   unsigned int size;
   unsigned int ea_low;
};
```

List elements are processed in the order specified. If the stall-and-notify flag is set in the transfer size, the MFC will halt processing of the DMA list until the SPU program has cleared the *DMA list stall event* from the external event status (channel 0). This gives programs an opportunity to modify individual list elements just before they are processed by the MFC.

The low order 32-bits of the EA is specified for each element in the list, but the LA involved in the transfer is only specified once, in the primary list command itself. The LA is internally incremented based on the amount of data transferred by each element in the list. However, due to alignment restrictions, if the LA does not begin on a 16-byte boundary for a list element transfer, the hardware will automatically increment the LA to the next 16-byte boundary[1].

The EA specified within a list element is relative to the 4GB area defined by the upper 32-bits of the EA specified in the primary DMA list command. While DMA list starting addresses are relative to the single 4GB area, transfers within a list element can cross the 4GB boundary.

Figure: DMA List Issue (SPU-C, 32-bit EA's)

```
#include <bpa_mfc.h>

struct spu_dma_list_elem list[NR_ELEMENTS1]
                 __attribute__ ((aligned (82)));
void *la;
unsigned int size3 =
   sizeof(struct spu_dma_list_elem)*NR_ELEMENTS;

/* List transfer using 32-bit EA */
spu_mfcdma32(la, (unsigned int) list, size,
             tag_id, MFC_GETL_CMD);
```

1. The maximum number of elements in an DMA list is 2048.
2. Storage for DMA lists must be aligined on an 8-byte boundary.
3. The transfer size of the base DMA list command is equal to the number of elements in the list times the size of the DMA list element structure (8-bytes)

Figure: DMA List Issue (SPU-C, 64-bit EA's)

```
union {
   unsigned long long ull;
   unsigned int ui[2];
} ea;

/* List transfer using 64-bit EA */
spu_mfcdma64(la, ea.ui[0], (unsigned int) list,
             size, tag_id, MFC_PUTLF_CMD);
```

7 Overlapping DMA and Computation

In this section we outline basic procedures for overlapping DMA transfers with computation.

1. This condition will only occur if transfer sizes less than 16-bytes are used. List elements having transfer sizes less than 16-bytes will utilize a local store offset within the current quadword (16-bytes) defined by the least significant 4-bits of the EA.

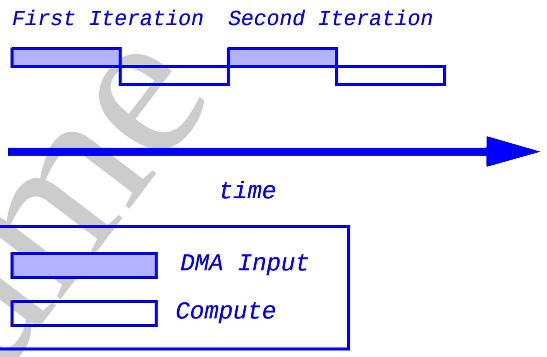
7.1 Double Buffering

Consider a simple SPU program that performs the following steps:

1. DMA data from EA to LS buffer ***B***.
2. Wait for the transfer to complete.
3. Compute on data in ***B***.
4. Repeat as necessary.

If considered as a time graph, the control flow for this program might look something like:

Figure: Serial Computation & Transfer



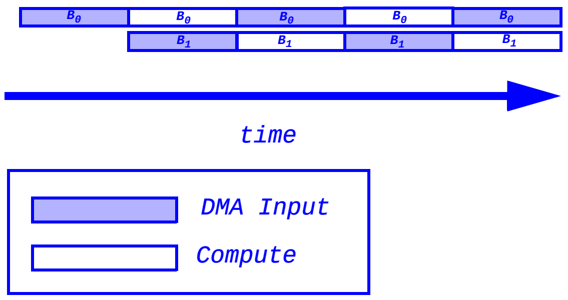
This sequence has no overlap between computation and data transfer. If the program is known to iterate more than once, then performance could be improved by allocating two LS buffers B_0 and B_1, and overlapping computation on one buffer with data transfer for the next. This technique is frequently referred to as *double buffering*.

The modified sequence is:

1. DMA data from EA to LS buffer B_0.
2. Wait for transfer of B_0 to complete.
3. DMA data from EA to LS buffer B_1.
4. Compute on data in B_0.
5. Wait for transfer of B_1 to complete.
6. DMA data from EA to LS buffer B_0.
7. Compute on data in B_1.
8. Repeat steps 2-7 as necessary.

As the new time graph illustrates, computation and data transfers are parallelized once the first DMA for B_0 completes.

Figure: Double Buffered Transfer



Double buffering may be achieved on the SPU by applying tag id 0 for all transfers of $\boldsymbol{B_0}$ (steps 1 & 6), and likewise applying tag identifier 1 for all transfers of $\boldsymbol{B_1}$ (step 3).

The key to the double buffering scheme is properly setting the tag group mask such that the program only waits for the particular buffer of interest. To wait for $\boldsymbol{B_0}$, set the tag group mask to include tag id 0, and request conditional tag status update for *all* tags in the mask (step 2). Likewise for $\boldsymbol{B_1}$, set the tag group mask to include tag id 1, and again request conditional tag status update for *all* tags in the mask (step 5).

Figure: Initiate Buffer Transfer (SPU-C)

```
#include <bpa_mfc.h>

typedef union {
   unsigned long long ull;
   unsigned int ui[2];
} addr64;

void *B[2]; /* Pointers to LS Buffers */

/* Initiate transfer using LS buffer B[i] */
static inline void xfer(addr64 ea,
                        unsigned int size,
                        unsigned int i)
{
   spu_mfcdma64(B[i], ea.ui[0], ea.ui[1],
               size, i, MFC_GET_CMD);
}
```

Figure: Wait for Buffer Transfer Complete (SPU-C)

```
/* Wait for B[i] transfer to complete. */
static inline void wait(unsigned int i)
{
   unsigned int tag_mask = (1 << i);

   spu_writech(MFC_WR_TAG_MASK, tag_mask);
   spu_mfcstat(MFC_TAG_UPDATE_ALL);
}
```

7.2 Multi-Buffering

Double buffering can be extended to support more than two LS buffers, provided that enough memory space is available in LS. When more than two buffers are involved, the technique is frequently referred to as *multi-buffering*.

Building on the concepts from the previous section, a basic recipe for multi-buffered data transfers on the SPU is as follows:

1. Allocate multiple LS buffers $\boldsymbol{B_0}$-$\boldsymbol{B_n}$.
2. Initiate transfers for buffers $\boldsymbol{B_0}$-$\boldsymbol{B_n}$. For each buffer $\boldsymbol{B_i}$, apply tag id $\boldsymbol{i}$ to transfers involving that buffer.
3. Beginning with $\boldsymbol{B_0}$ and moving through each of the buffers in round robin fashion:
   – set tag group mask to include only tag $\boldsymbol{i}$, and request conditional tag status update for *all* tags to complete
   – compute on $\boldsymbol{B_i}$
   – initiate next transfer on $\boldsymbol{B_i}$

This recipe waits for and processes each $\boldsymbol{B_i}$ in round robin order, regardless of when the transfers complete with respect to one another. In this sense we can say that this recipe uses a *strongly ordered* transfer model. An alternative would be to set the tag group mask to include all tag identifiers for $\boldsymbol{B_0}$-$\boldsymbol{B_n}$, and then request conditional tag status update for *any* tags that have completed (and subsequently process the first buffer that completes). This would represent a *weakly ordered* transfer model. While the weakly ordered model may have advantages in theory, it introduces complexity and is unlikely to pay off in practice.

Of course either incoming or outgoing data transfers can cause bottlenecks for performance. It may therefore be necessary to implement buffering schemes for both input and output. The techniques mentioned in these sections can be applied to either type of transfer.

7.3 Shared I/O Buffers

In this section we examine a particular case of multi-buffered input and output, where the storage for the incoming and outgoing buffers are shared. The advantage of using shared input and output buffers is that LS memory requirements are reduced.

Consider the previous double-buffered SPU program, but include an additional output stage to the sequence, as follows:

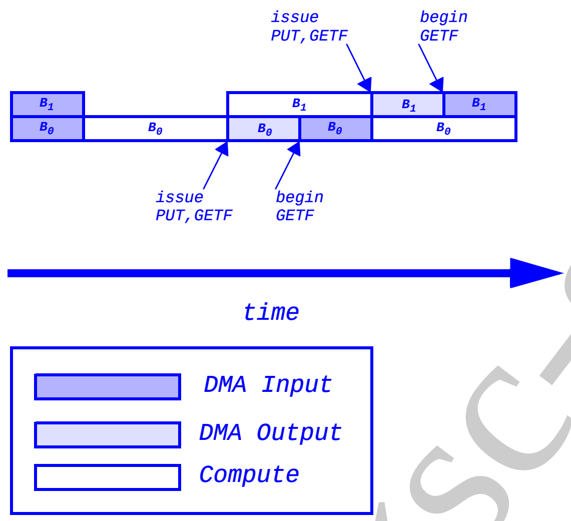
1. DMA data from EA to LS buffer $\boldsymbol{B_0}$.
2. DMA data from EA to LS buffer $\boldsymbol{B_1}$.
3. Wait for transfer of $\boldsymbol{B_0}$ to complete.
4. Compute on data in $\boldsymbol{B_0}$ and store results back to $\boldsymbol{B_0}$.
5. DMA data from LS buffer $\boldsymbol{B_0}$ to EA.
6. Wait for transfer of $\boldsymbol{B_0}$ to complete.
7. DMA data from EA to LS buffer $\boldsymbol{B_0}$.
8. Wait for transfer of $\boldsymbol{B_1}$ to complete.

9. Compute on data in B_1 and store results back to B_1.
10. DMA data from LS buffer B_1 to EA.
11. Wait for transfer of B_1 to complete[1].
12. Repeat steps 2-11 as necessary.

This sequence would require the program to stall waiting for outbound transfers to complete (steps 6 & 11) before new incoming transfers can be initiated (steps 2 & 7).

Fortunately, the MFC supports *fenced* variants of all the *GET* and *PUT* commands. The fence attribute causes a command to be locally ordered with respect to all previously issued commands within the same tag group. By using fenced *GET* commands, we no longer need to wait for outbound transfers to complete (steps 6 & 11), and parallelism is maintained.

Figure: Shared I/O Buffers with Fenced GET



7.4 Guidelines for Data Buffering

The motivation behind a data buffering scheme is of course to maximize the percentage of time spent in the compute phase of a program. Conversely, we could also say that the goal is to minimize the percentage of time spent waiting for DMA transfers to complete.

Let τ_c represent the time required to process a buffer B, and let τ_t represent the time required to transfer that buffer. In general, the higher the ratio τ_t/τ_c, the more performance benefit an application will realize from a DMA buffering scheme.

The following table lists basic guidelines to consider for DMA buffering schemes.

Figure: Data Buffering Recommendations

τ_t/τ_c	**multi-buffering benefit**	**# buffers (input OR output)**	**# buffers (input AND output)**
< 0.1	very poor	1	1
0.5	moderate	2	2
1.0	good	2	4
2.0	very good	3	8
4.0	very good	5	16
8.0	very good	9	32

References

1. *Broadband Processor Architecture, Books I-III*
STI Design Center
2. *PowerPC AS Architecture, Books I-III*
IBM
3. *SPU C/C++ Language Extensions, version 1.4*
BPA JSRE Series
4. *SPU Assembly Language Specification, version 0.4*
BPA JSRE Series

1. Eliminate steps 6 & 11 by using fenced GET commands.