PS3 Programming

Week 3. More DMA techniques

Chap 7 and chap 12

Outline

- Multi-buffered DMA
- DMA request lists
- DMA synchronization
- Homework

MULTI-BUFFERED DMA

A standard SPU's code

```
/* Read unprocessed data from main memory */
mfc_get(buff, argp,  sizeof(buff), TAG, 0, 0);
mfc_write_tag_mask(1<<TAG);
mfc_read_tag_status_all();

/* Process the data */
....

/* Write the processed data to main memory */
mfc_put(buff, argp,  sizeof(buff), TAG, 0, 0);
mfc_write_tag_mask(1<<TAG);
mfc_read_tag_status_all();
```

Synchronized data access

- A standard process
  - SPUs read data from main memory into LS
  - SPUs process data (need to wait read data)
  - SPU write data from LS to main memory
- There is no multi-threading in SPU to overlap the communication/computation.
  - SPUs are idle when waiting data
  - Use multi-buffer to hide the communication time

Double buffer

- Create a buffer as large as the incoming data

```
/* The buffer is twice the size of the data */
  vector unsigned int buff[SIZE*2]
    __attribute__ ((aligned(128)));
  unsigned short block_size = sizeof(buff)/2;
```

- Let SPUs process the data in one half of the buffer and let MFC get the data in another half of the buffer simultaneously.
- Change the buffer alternatively

```
/* Fill low half with unprocessed data */
 mfc_get(buff, argp, block_size, 0, 0, 0);

 for(i=1; i<8; i++) {
   /* Fill new buffer with unprocessed data  */
   mfc_get(buff +(i&1)*SIZE, argp+i*block_size,
                block_size, i&1, 0, 0);

   /* Wait for old buffer to fill/empty */
   mfc_write_tag_mask(1<<(1-(i&1)));
   mfc_read_tag_status_all();
   /* Process data in old buffer */
  ....

   /* Write data in old buffer to memory */
   mfc_put(buff+(1-(i&1))*SIZE, argp+(i-1)*block_size,
                 block_size, 1-(i&1), 0, 0);
 }
```

DMA REQUEST LISTS

Communication cost

- Communication cost includes
  - # words moved / bandwidth
  - # messages * latency
- Exponentially growing gaps between
  - Flop/s << 1/Network BW << Network Latency
    - Improving 59%/year vs 26%/year vs 15%/year
  - Flop/s << 1/Memory BW << Memory Latency
    - Improving 59%/year vs 23%/year vs 5.5%/year

Vectored I/O

- Put multiple memory access calls into one
  - Reduce the latency of I/O
- DMA request list (for scatter/gather)
  - Can hold up to 2,048 DMA transfers
  - Maximum data movement is 2048x16K=32MB
  - Cannot call mcf_get and mcf_put in the same list
- Two-step process
  - Create a list element data structure
  - Call a DMA list function

DMA list elements

```
typedef struct mfc_list_element{
   unsigned int notify   : 1;
   unsigned int reserved : 16;
   unsigned int size     : 15;
   unsigned int eal      : 32;
} mfc_list_element_t;
```

- Defined in spu_mfcio.h
  - Lower half effective address (eal): more latter
  - The size of data to transfer (<16KB)
  - If notify==1, the list call stops.

DMA list functions

- Six functions: `mfc_getl, mfc_putl, mfc_getlf, mfc_putlf, mfc_getlb, mfc_putlb`
- Arguments
  - volatile void *ls: LS address of the data
  - unsigned long long ea: the effective address (EA)
  - volatile mfc_list_element_t *list: array of list elm
  - unsigned int size: size of list
  - unsigned int tag, tid, rid: same as they are in mfc_get / mfc_put

Some pitfalls

- Only the most significant 32 bits of ea are used. The least significant 32 bits are provided by the lea in list elements.
  - Use mfc_ea2l, mfc_ea2h, mfc_hl2ea, and mfc_ceil128 to manipulate effective address
- The argument size is the size of *list, not the size of data. The size of data is specified by the size in list elements.

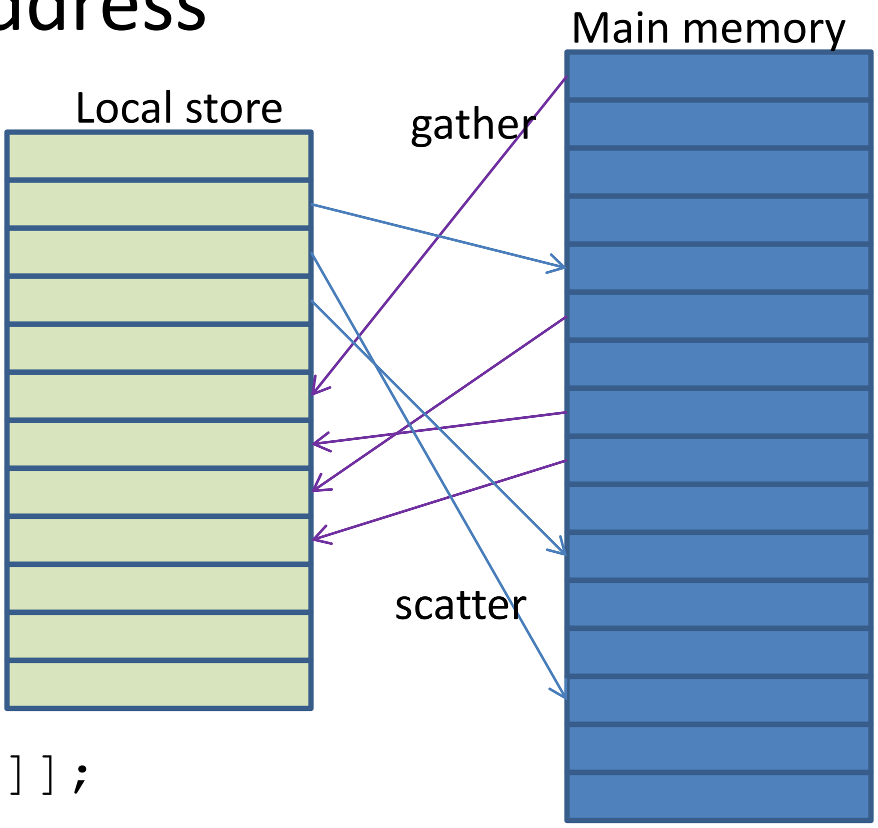
Scatter / gather I/O

- There is only one LS address
  - The initial address of data transfer
- Gather:

```
for(i=0; i<N; ++i)
    A[i]=B[i]+C[D[i]];
```

- Scatter:

```
for(i=0;i<N;++i) ++A[B[i]];
```



Example: spu's code

```
#include <spu_mfcio.h>
#include <spu_intrinsics.h>

#define SIZE 4096
#define TAG 3

unsigned int hold_array[SIZE*4]   __attribute__ ((aligned (128)));

int main(vector unsigned long long arg1,
      vector unsigned long long arg2,
      vector unsigned long long arg3) {

   unsigned long long get_addr, put_addr[4];
   int i;
```

More on this later

Get list data

```
/* Retrieve the five addresses from the input parameters */
get_addr = spu_extract(arg1, 0);

/* Create list elements for mfc_getl */
mfc_list_element_t get_element[4];
for (i=0; i<4; i++) {
  get_element[i].size = SIZE*sizeof(unsigned int);
  get_element[i].eal = mfc_ea2l(get_addr) + i*SIZE*sizeof(unsigned int);
}

/* Transfer data into LS */
mfc_getlb(hold_array, get_addr, get_element,
            sizeof(get_element), TAG, 0, 0);
mfc_write_tag_mask(1<<TAG);
mfc_read_tag_status_all();
```

Put list data

```
/* Retrieve the addresses from the input parameters */
  put_addr[0] = spu_extract(arg1, 1);
  put_addr[1] = spu_extract(arg2, 0);
  put_addr[2] = spu_extract(arg2, 1);
  put_addr[3] = spu_extract(arg3, 0);

/* Create list elements for mfc_putl */
  mfc_list_element_t put_element[4];
  for (i=0; i<4; i++) {
    put_element[i].size = SIZE*sizeof(unsigned int);
    put_element[i].eal = mfc_ea2l(put_addr[i]);
  }
  /* Transfer data out of LS */
  mfc_putl(hold_array, put_addr[0], put_element,
              sizeof(put_element), TAG, 0, 0);
  mfc_write_tag_mask(1<<TAG);
  mfc_read_tag_status_all();
}
```

PPU’s code

```
#include <stdio.h>
#include <stdlib.h>
#include <libspe2.h>
#include <ppu_intrinsics.h>

#define SIZE 4096
/* Program handle representing the SPU object */
extern spe_program_handle_t spu_dmalist;

int main() {
  int i, retval;   spe_context_ptr_t spe;
  unsigned entry = SPE_DEFAULT_ENTRY;   spe_stop_info_t stop_info;

  /* Array to be transfered into the SPU's LS */
  unsigned int large_array[SIZE*4] __attribute__ ((aligned (128)));
```

```
/* Arrays to hold data transferred out of the LS */
 unsigned int small_1[SIZE] __attribute__ ((aligned (128)));
 unsigned int small_2[SIZE] __attribute__ ((aligned (128)));
 unsigned int small_3[SIZE] __attribute__ ((aligned (128)));
 unsigned int small_4[SIZE] __attribute__ ((aligned (128)));

 /* Fill the array with whole numbers */
 for(i=0; i<SIZE*4; i++)   large_array[i] = i;

 /* Initialize the array of array addresses */
 unsigned long long control_block[6];
 control_block[0] = (unsigned long long)large_array;
 control_block[1] = (unsigned long long)small_1;
 control_block[2] = (unsigned long long)small_2;
 control_block[3] = (unsigned long long)small_3;
 control_block[4] = (unsigned long long)small_4;
 control_block[5] = (unsigned long long)NULL;
```

```
/* Create the SPE Context */
 spe = spe_context_create(0, NULL);
 if (!spe) {
   perror("spe_context_create");   exit(1);
 }

 /* Load the program into the context */
 retval = spe_program_load(spe, &spu_dmalist);
 if (retval) {
   perror("spe_program_load");   exit(1);
 }

 /* Run the program inside the context */
 retval = spe_context_run(spe, &entry, SPE_RUN_USER_REGS,
           control_block, NULL, &stop_info);
 if (retval < 0) {
   perror("spe_context_run");   exit(1);
 }
```

```
/* Deallocate the context */
  retval = spe_context_destroy(spe);
  if (retval) {
    perror("spe_context_destroy");      exit(1);
  }

  /* Check that the received data is valid */
  int test = 1;
  for(i=0; i<SIZE; i++) {
     if(small_1[i] != i)              test = 0;
     if(small_2[i] != SIZE+i)         test = 0;
     if(small_3[i] != 2*SIZE+i)   test = 0;
     if(small_4[i] != 3*SIZE+i)   test = 0;
  }
  if(test)    printf("DMA Passed.\n");
  else        printf("DMA Failed.\n");
  return 0;
}
```

SYNCHRONIZATION

Synchronization library (libsync)

- Functions are divided into five categories
  1. Atomic operations: uninterruptable operations
     - atomic_read, atomic_set, atomic_inc, …
  2. Mutexes
  3. Reader/writer locks
  4. Condition variable operations
  5. Completion operations
     - A special condition variable that signals others when the thread is finished.

Mutex (mutual execlusive)

- Protect critical regions

```
mutex_lock(a)
// critical region (CR)
// access mutual exclusive resources.
mutex_unlock(a)
```

Spinning lock
Wait until a==0
And then set a=1

- Functions

```
mutex_init(mutex_ea_t lock)
mutex_lock(mutex_ea_t lock)
mutex_trylock(mutex_ea_t lock)
mutex_unlock(mutex_ea_t lock)
```

Reader/writer locks

- A set of more sophisticated mutex functions
  - Mutex only allows one thread to enter the CR
  - Sometimes, it is ok for more than 1 threads to access data, like read. Mutual exclusion only happens when there is a write operation.
- Rules: suppose the lock is “a”. (typed `mutex_ea_t`)
  - If a!=0, the write_lock causes a spinning lock
    - After it sees a==0, it sets a = -1
  - If a==-1, read_lock causes spinning lock
    - After it sees a!=-1, it sets a = a+1

Condition variables

- The goal of a condition variable is to have all threads wait until a condition happens.
  - The condition is triggered by other threads.
- Condition variables are always associated with mutexes to avoid race conditions.

```
mutex_lock(a)
cond_wait(cond,a)
mutex_unlock(a)
```

It will release "a". Until the condition is satisfied, it re-locks "a".

```
mutex_lock(a)
cond_signal(cond)
mutex_unlock(a)
```

Or use `cond_croadcast`

SPU’s cashier code

```
#include <spu_mfcio.h>
#include <spu_intrinsics.h>
#include <libsync.h>
#define TAG 3

/* SPU initialization data */
typedef struct _control_block {
  cond_ea_t cashier;
  mutex_ea_t cashier_mutex, served_mutex;
  unsigned long long served_addr;
} control_block;

control_block cb __attribute__ ((aligned (128)));

int main(unsigned long long speid,  unsigned long long argp,
          unsigned long long envp) {
```

```c
/* Get the control block from main memory */
  mfc_get(&cb, argp, sizeof(cb), TAG, 0, 0);
  mfc_write_tag_mask(1<<TAG);   mfc_read_tag_status_all();

  /* Enter the store: get lock to wait for cashier */
  mutex_lock(cb.cashier_mutex);
  /* Wait for cashier */
  cond_wait(cb.cashier, cb.cashier_mutex);
  /* Allow others to wait for the cashier */
  mutex_unlock(cb.cashier_mutex);

  /* Leave the store: get lock to increment num_served */
  mutex_lock(cb.served_mutex);
  /* Increment the number of customers served */
  atomic_inc((atomic_ea_t)cb.served_addr);
  printf("Thread %llu incremented num_served to %u\n",
    speid, atomic_read((atomic_ea_t)cb.served_addr));
  /* Allow others to access num_served */
  mutex_unlock(cb.served_mutex);
  return 0;
}
```

PPU’s cashier code

```
/* SPU initialization data */
typedef struct _control_block {
  cond_ea_t cashier;
  mutex_ea_t cashier_mutex, served_mutex;
  unsigned long long served_addr;
} control_block;

/* SPU program handle */
extern spe_program_handle_t spu_cashier;
ppu_pthread_data_t data[6];
control_block cb __attribute__ ((aligned (128)));

/* Declare variables */
volatile int cashier_var __attribute__ ((aligned (128)));
volatile int cashier_mutex_var __attribute__ ((aligned (128)));
volatile int served_mutex_var __attribute__ ((aligned (128)));
volatile int num_served __attribute__ ((aligned (128)));
```

Same as SPU’s

```
/* The data sent to the pthread */
typedef struct ppu_pthread_data {
    spe_context_ptr_t speid;
    pthread_t pthread;
    void *argp;
} ppu_pthread_data_t;

/* The function executed in the pthread */
void *ppu_pthread_function(void *arg) {
   ppu_pthread_data_t *data = (ppu_pthread_data_t *)arg;
   int retval;
   unsigned int entry = SPE_DEFAULT_ENTRY;
   if ((retval = spe_context_run(data->speid,  &entry, 0, data->argp, NULL,
NULL)) < 0) {
      perror("spe_context_run");     exit (1);
   }
   pthread_exit(NULL);
}
```

```
int main(int argc, char **argv) {
  int i, retval, spus;

  /* Create condition variable, mutexes, pointer */
  cb.cashier = (cond_ea_t)&cashier_var;
  cb.cashier_mutex = (mutex_ea_t)&cashier_mutex_var;
  cb.served_mutex = (mutex_ea_t)&served_mutex_var;
  cb.served_addr = (unsigned long long)&num_served;

  /* Initialize condition variable, mutexes */
  cond_init(cb.cashier);
  mutex_init(cb.cashier_mutex);
  cond_init(cb.served_mutex);
  num_served = 0;

  /* Determine number of available SPUs */
  spus = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, 0);
```

```
/* Create contexts and threads */
for (i=0; i<spus; i++) {     /* Create context */
    if ((data[i].speid = spe_context_create(0, NULL)) == NULL) {
      perror("spe_context_create");     exit(1);
    }

    /* Load program into the context */
    if ((retval = spe_program_load(data[i].speid,   &spu_cashier)) != 0) {
      perror("spe_program_load");     exit (1);
    }
   data[i].argp = &cb;

    /* Create thread */
    if ((retval = pthread_create(&data[i].pthread,
                          NULL, &ppu_pthread_function, &data[i])) != 0) {
      perror("pthread_create");   exit (1);
    }
 }
```

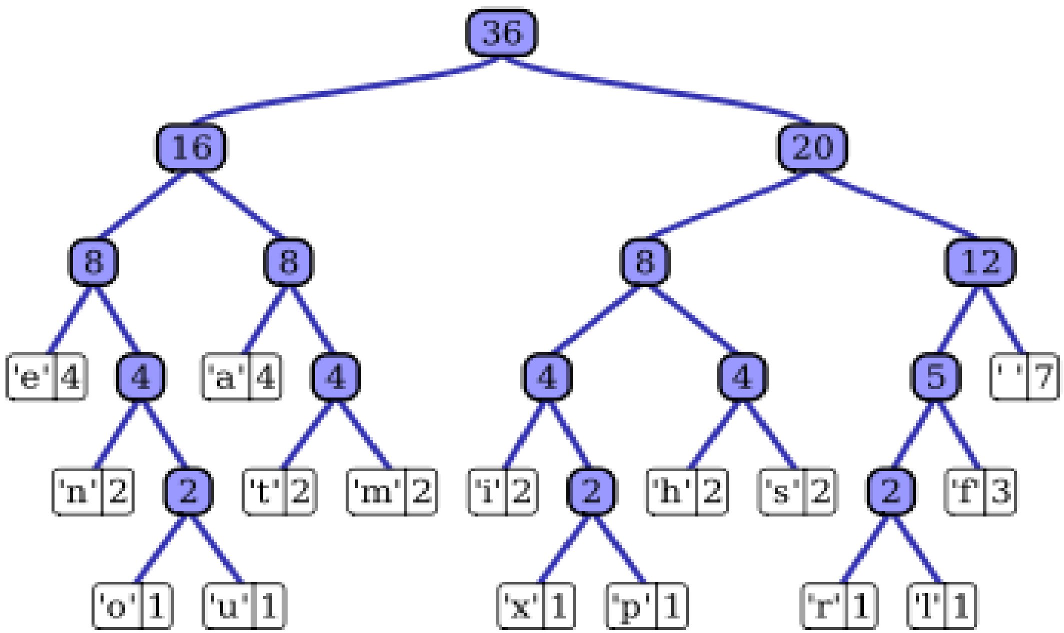
```
int count = 0;
while (count < spus) {
  mutex_lock(cb.cashier_mutex);
   cond_signal(cb.cashier);
   count = atomic_read((atomic_ea_t)cb.served_addr);
   mutex_unlock(cb.cashier_mutex);
 }

for (i = 0; i < spus; i++) {   /* Wait for the threads */
   if ((retval = pthread_join   (data[i].pthread, NULL)) != 0) {
      perror("pthread_join");  exit (1);
   }

   /* Deallocate the contexts */
   if ((retval = spe_context_destroy (data[i].speid)) != 0) {
      perror("spe_context_destroy");     exit (1);
   }
 }
return 0;
```

HOMEWORK

Huffman coding



- Given a set of symbols and their counts, find a prefix-free binary code with minimum expected codeword length

Char	Freq	Code
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000
u	1	00111
x	1	10010

Huffman coding for compression

1. Read in a file (binary or text)
2. Count its char frequency
3. Build the Huffman tree according to the char frequency
   – Use priority queue (heap sort)
4. Write out the Huffman coded file
   – The code table and the compressed data
5. Design the decompression function (optional)

Homework

- Implement a sequential Huffman compression and decompression codes using PPU
  - Use as many vectored statements as possible
  - You don’t need to implement a priority queue, but it would be a good practice
  - You can designed your own compression file format
- Use SPUs to help