Freescale Semiconductor
Application Note

Document Number:AN4749
Rev 0, 07/2013

Configuring CoreNet Platform Cache (CPC) as SRAM

For Use by Linux Applications

Contents

1 Introduction

This document provides, with examples, software methods to configure the CoreNet Platform Cache (CPC) as very high performance SRAM for use by Linux applications.

The CPC is a CoreNet-compliant target device that can serve as a general purpose write-back, an I/O stash, a memory-mapped SRAM device, or any combination of these functions. As a general purpose cache, the CPC manages allocations and victimizations to reduce read latency and increase bandwidth for accesses to backing store (DRAM). As an I/O stash, the CPC can accept and allocate writes from an I/O device in order to reduce latency and improve bandwidth for multiple read operations to the same address. As an SRAM device, the CPC acts as a low-latency, high-bandwidth memory that occupies a programmable address range.

Below, illustrative examples from the P4080 are discussed which demonstrate how to configure the CPC in both hardware and software. Finally, examples show methods for measuring performance benchmarking results.

Readers should be familiar with the P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual.

2 Configuring the CPC SRAM in hardware

At a high level, the steps required to configure the CPC hardware are shown below:

Table 1. USDPAA offloading applications

Step	Description
1	Use the SRAM mode registers to configure the CPC to use some of its (32) ways as a direct-mapped SRAM memory.
2	Enable the CPC's SRAM mode with the CPCSRCR0[SRAMEN] bit.
3	Control the size of the SRAM address space with the CPCSRCR0[SRAMSZ] bits.
4	Control the base address of the SRAM address space with the CPCSRCR0[SRBARL] and CPCSRCR1[SRBARU] bits.
5	Interleave SRAM address space on the same address boundary by setting the CPCSRCR0[INTLVEN] bit if DRAM addresses are interleaved using the DDR controller configuration registers. (If DDR addresses are not interleaved, SRAM address space cannot be interleaved.)

NOTE

SRAM interleaving requires that each CPC instance in the system that participates in interleaving be configured to use the same SRAM size and base address. Enabling SRAM interleaving effectively doubles the total SRAM address space.

2.1 Modifying the register settings for CPC SRAM mode

Change the CPC SRAM control register 1 (CPCSRCR1) to modify the upper base address when the CPC is used as SRAM. The least significant 16 bits defines the SRAM physical address 16:31. Note that for each CPU core, it is 64 bits address width. But for SRAM, it can only be set up to 48 bits (`0xffff_ffff_ffff`)

The CPC SRAM control register 0 (CPCSRCR0), configures the CPC when it is used as SRAM. The most significant 16 bits defines the SRAM physical address 32:48. This means that the SRAM base address can only be set at 32K boundary. Also, this register is used to configure SRAM size and interleaving mode. The least significant bit of the register is used to enable or disable SRAM mode.

In addition to the registers described above, the SRAM/CACHE mode must be enabled using the CPC configuration and status register (CPCCSR0).

3 Modifying the Software configuration

The CPC SRAM is designed for cases in which efficiency is critical. For example, for high-speed data loading and data storage and transfer. In the ideal case, SRAM throughput can be up to:

```
CoreNet frequency * CoreNet data width
```

For the P4080 SoC, this works out to 800MHz * 128 bits/sec. That said, real-world performance is limited by software and DMA performance.

3.1 Invoking the U-boot configuration

The following shows how to use U-boot to enable the CPC to speed up image loads, decompression, etc. For Linux applications that require SRAM, one can configure the CPC as well as SRAM and as cache in U-boot.

Example U-boot SRAM configuration code is shown here:

```
/* Configure the SRAM upper base address */
out_be32(&cpc-> cpcsrcr1, CPC_SRCR1_SRBARU(SRAM_BASE_ADDRESS));

/* Configure the SRAM lower base address, SRAM size and enable SRAM mode */
out_be32(&cpc->cpcsrcr0,
    CPC_SRCR0_SRAMEN |
    CPC_SRCR0_SRAMSZ_32_WAY |
    CPC_SRCR0_SRBARL(SRAM_BASE_ADDRESS));

/*Enable the SRAM CS */
out_be32(&cpc->cpccsr0, CPC_CSR0_CE);

/* Read the configuration to let it work */
in_be32(&cpc-> cpccsr0);

/* Set the SRAM access window in memory address space */
law_idx = set_next_law(SRAM_BASE_ADDRESS, LAW_SIZE_1M,  LAW_TRGT_IF_DDR_1);

/* Set a TLB1 entry for SRAM , it is not necessary in U-boot, just for test.
SRAM TLB1 entry will be set in Linux eventually  */
tlb_idx = find_free_tlbcam();
set_tlb(1, SRAM_VIRTUAL_BASE_ADDRESS, (u64) SRAM_BASE_ADDRESS,
    MAS3_SX | MAS3_SW | MAS3_SR, 0, 0, tlb_idx, BOOKE_PAGESZ_1M, 1);

/* Set the other CPC chip as cache */
cpc++
out_be32(&cpc->cpccsr0, CPC_CSR0_CE | CPC_CSR0_PE);
in_be32(&cpc->cpccsr0);
```

In the U-boot code above, SRAM is mapped to SRAM_BASE_ADDRESS, which is the physical base address defined by the user. It is suggested that the SRAM address not overlap with the DDR controller's region or any other devices' I/O region. U-boot can configure the SRAM base address and size using the **hwconfig** parameters.

3.2 Linux kernel configuration

The SRAM's physical base address and size configured in U-boot must be passed to the Linux kernel as parameters. The sections that follow explain how the kernel allows applications to access SRAM.

3.2.1 mmap interface callback

The QorIQ DPAA SDK includes a Linux SRAM device driver that provides an interface that lets application use the SRAM. An application calls this drivers' `mmap` interface to access the SRAM address space. In the `mmap` callback function, the function `remap_pfn_range` is called for the current user process to map its virtual address to SRAM physical address. However, translation lookaside buffer (TLB) entries for this mapping are not created.

```
static int sram_mmap(struct file *file, struct vm_area_struct *vma)
{
    size_t size = vma->vm_end - vma->vm_start;
```

```
        if (remap_pfn_range(vma, vma->vm_start, (sram_alloc_addr >> PAGE_SHIFT),
            size, vma->vm_page_prot))
            {
                printk("sram mmap error\r\n");
                return -EAGAIN;
            }
    }
```

3.2.2 TLB miss exception handling

An application can get a virtual address mapped to SRAM using the `mmap` system call. However, as stated above, a TLB entry for the mapping is not set up. Therefore, when you access a virtual address returned by `mmap`, a TLB miss exception is asserted.

In the TLB miss exception interrupt service routine (ISR), the function shown below is called to set up a TLB1 entry for this virtual address to physical address translation.

```
static inline void hook_sramcpc_tlb(struct vm_area_struct *vma,
            unsigned long address, pte_t *ptep)
{
    unsigned long pfn = pte_pfn(*ptep);
    if ((pfn < (sramcpc_pfn_start + sramcpc_pfn_len)) &&
        (pfn >= sramcpc_pfn_start))
        {
            unsigned long va = address & ~(sramcpc_phys_size - 1);
            flush_tlb_mm(vma->vm_mm);
            settlbcam(usdpaa_tlbcam_index, va,
                    sramcpc_phys_start, sramcpc_phys_size,
                    pte_val(*ptep), 0);
        }
}
```

Consider an SMP Linux kernel running on an eight CPU core SoC in which each core has its own TLB and is running a single process. When a given process accesses SRAM, an SRAM address translation TLB1 entry is set up by the TLB miss exception ISR of the core on which this process is running. If a second process is spawned on this core and this process accesses SRAM, a TLB miss exception is **not** asserted. Instead, the address translation uses the TLB entry created when the first process accessed SRAM.

3.2.3 SRAM memory management

All processes share the same SRAM space. Therefore we must break the SRAM into partitions so one process cannot overwrite data written by another.

A 1 MB SRAM can be broken into 256 4KB blocks. We create a memory map to record memory use. When an application calls the `mmap` callback to allocate memory in the SRAM, this map is checked to determine if there is a sufficient contiguous memory to satisfy the request. If there is, the physical beginning address of the contiguous memory (`sram_alloc_addr`) is mapped to the application's virtual address. The virtual address returned to the application by `mmap` is BASE_VIRTUAL_ADDRESS + `offset` where BASE_VIRTUAL_ADDRESS is 0 and `offset` is the offset to the SRAM base address of the allocated blocks. Otherwise, an error is returned to notify the application that the operation failed.

```
vma->vm_star= BASE_VIRTUAL_ADDRESS+4*1024*begin_blocks;

vma->vm_end= vma->vm_star+4*1024*allocated_blocks;

remap_pfn_range(vma, vma->vm_start, (sram_alloc_addr >> PAGE_SHIFT),
        size, vma->vm_page_prot);
```

A process is allowed to acquire memory from the SRAM no more than ten times using the mmap interface. Further, before calling the mmap, the application should call ioctl to notify the kernel driver of the alloc ID of the SRAM memory block being acquired. When an SRAM memory block is no longer needed or before a process terminates, the process must call ioctl to free the memory, otherwise a memory leak results.

The process ID and alloc ID are used to record memory usage in memory map.

3.3 Using SRAM from a Linux application

An application can access the SRAM via the normal driver interface, the mmap system call. When a block of SRAM is no longer needed, the application must call ioctl to free the memory. An application can also use ioctl to get the SRAM physical address for DMA operations.

Shown below is typical code that manipulates SRAM:

```
unsigned long long  allocID=1;
volatile unsigned int* temp_addr;
fd = open(CPC_SRAM_PATH, O_RDWR);
ret = ioctl(fd, CPCSRAM_IOCTL_ALLOC, &allocID);
virt = mmap(0, 200*1024, PROT_READ | PROT_WRITE,MAP_SHARED | MAP_FIXED, fd, 0);
temp_addr = (volatile unsigned int*)virt;
*temp_addr= *temp_addr + 1; //access the SRAM
allocID = 2;
ret = ioctl(fd, CPCSRAM_IOCTL_ALLOC, &allocID);
virt = mmap(0, 400*1024, PROT_READ | PROT_WRITE,MAP_SHARED | MAP_FIXED, fd, 0);
temp_addr = (volatile unsigned int*)virt;
*temp_addr = *temp_addr+10;//access the SRAM
allocID = 1;
ret = ioctl(fd, CPCSRAM_IOCTL_FREE, &allocID);
allocID = 2;
ret = ioctl(fd, CPCSRAM_IOCTL_FREE, &allocID);
close(fd);
```

4 Understanding the SRAM performance benchmark

In the ideal case, for a CoreNet running at a frequency of 800MHz with a data width of 128 bits, SRAM throughput is:

```
800 MHz * 128 bits/second
```

However, this calculation does not take into consideration SRAM access wait-states. The DDR controller's working frequency is also 800 MHz, but considering the latency of preparing for 64-byte burst accesses and page open, DDR throughput is about half that of the SRAM throughput.

4.1 Software benchmark

We cannot calculate SRAM bandwidth using software because it is extremely difficult to exclude the time consumed executing core instructions. However, it is easy to compare the throughputs of DDR memory and CPC SRAM. The code shown here makes this comparison:

```
dcache_disable(); // L1 & L2 data cache disable for DDR and SRAM benchmark
atbl0 = mfspr(526); atbu0 = mfspr(527);
time_begin = (u64)atbu0;
time_begin <<= 32;
time_begin = time_begin | (u64)atbl0;
```

```
for( j = 0; j < 500; j++)//This loop is easy to test and compare
{
    temp=ddr_ptr;
    for( k = 0;k < (1024*256) / 16; k++){
              *(temp+0) = *(temp+1);
              *(temp+1) = *(temp+2);
              ……
              *(temp+14) = *(temp+15);
              *(temp+15) = *(temp+0);

                   temp=temp+16;
        } /* end inner loop */
} /* end outer loop */

atbl1 = mfspr(526); atbu1 = mfspr(527);
time_end = (u64)atbu1;
time_end <<= 32;
time_end = time_end | (u64)atbl1;
printf("soft DDR cycle expired:0x%llx\r\n", time_end-time_begin);
atbl0 = mfspr(526); atbu0=mfspr(527);
atbl0 = mfspr(526); atbu0=mfspr(527);
time_begin = (u64)atbu0;
time_begin <<= 32;
time_begin = time_begin | (u64)atbl0;
for( j = 0; j < 500; j++)
{
    temp=sram_ptr;
    for( k = 0; k < (1024 * 256) / 16; k++){
        *(temp + 0) = *(temp + 1);
        *(temp + 1) = *(temp + 2);
        ……
        *(temp + 14) = *(temp + 15);
        *(temp + 15) = **(temp + 0);

        temp=temp+16;
    } /* end inner loop */
} /* end outer loop */

atbl1 = mfspr(526); atbu1 = mfspr(527);
time_end = (u64)atbu1;
time_end <<= 32;
time_end = time_end | (u64)atbl1;
printf("soft SRAM cycle expired:0x%llx\r\n", time_end - time_begin);
```

The test code's output proves that the SRAM efficiency is about twice that of DDR memory.

```
soft DDR cycle expired: 0x23eea58e5
soft SRAM cycle expired: 0xfac014a2
```

4.2 DMA benchmark

We cannot exploit the SRAM's superior performance when used for DMA transfers. This is because the OCN which connects the DMA engine has limited bandwidth and creates a bottleneck. As the test results show, the DMA-to-SRAM transfer performance is about the same as the DMA-to-DMA transfer. This limits application of SRAM a great deal.

Here is the code for the DMA-to-SRAM and the DMA-to-DMA benchmarks:

```
out_dma32(&dma->satr, FSL_DMA_SATR_SREAD_SNOOP);
out_dma32(&dma->datr, FSL_DMA_DATR_DWRITE_SNOOP);
out_dma32(&dma->sr, ffff_ffffh); /* clear any errors */
dma_sync();
xfer_size = count; //MIN(FSL_DMA_MAX_SIZE, count);
out_dma32(&dma->dar, (u32) (dest & FFFF_FFFFh));
out_dma32(&dma->sar, (u32) (src & FFFF_FFFFh));
```

```
out_dma32(&dma->satr,(in_dma32(&dma->satr)&ffff_fc00) | (u32)((u64)src >> 32));
out_dma32(&dma->datr,in_dma32(&dma->datr) | (u32)((u64)dest >> 32));
out_dma32(&dma->bcr, xfer_size);
dma_sync();
out_dma32(&dma->mr, FSL_DMA_MR_DEFAULT|0002_8000h);
dma_sync();
atbl0=mfspr(526);atbu0=mfspr(527);
time_begin = (u64)atbu0;
time_begin <= 32;
   time_begin = time_begin | (u64)atbl0;
out_dma32(&dma->mr, 0800_0000h|FSL_DMA_MR_DEFAULT |
             FSL_DMA_MR_CS|0002_8000h);
dma_sync();

    do {
        status = in_dma32(&dma->sr);
    } while (status & FSL_DMA_SR_CB);

out_dma32(&dma->mr, in_dma32(&dma->mr) & ~FSL_DMA_MR_CS);
dma_sync();

if (status != 0){
        printf ("DMA Error: status = %x\n", status);
            return status;
}
atbl1=mfspr(526);atbu1=mfspr(527);
time_end = (u64)atbu1;
        time_end <<= 32;
        time_end = time_end | (u64)atbl1;
printf("HH: 00DMA test  time expired:0x%11x\n",time_end-time_begin);
```

5 Configuring a standalone Linux kernel.

A CPC SRAM configuration for a standalone Linux kernel that does not depend on U-boot configuration has not been implemented.

However, the sections that follow cover the basic concepts implementing a physical CPC SRAM configuration in a standalone Linux kernel.

5.1 Disable CPC configuration

To reconfigure the CPC as SRAM, the CPC configuration should first be disabled.

Table 2. Steps to disable CPC

Step	Action	Description
1	Clear all bits of `CPC ARn`	This prevents any new transactions from allocating CPC resources using a write operation.
2	Flush the CPC by performing a `dcbf` operation on each coherency granule mapped to the memory target (i.e. one `dcbf` for each 64 bytes.	To do this, first read 2MB of contiguous memory in a loop. When this operation completes, the CPC cache lines are filled with this memory region. Then, in a loop, perform a `dcbf` operation on every 64 bytes to cover this entire memory region, thereby flushing the CPC.
3	Clear all lock bits by setting CPCCSR0[CPCLFC].	-

Table continues on the next page...

Table 2. Steps to disable CPC (continued)

Step	Action	Description
4	Wait for the hardware to clear CPCCSR0[CPCLFC] using a polling loop.	-
5	Disable the CPC by clearing CPCCSR0[CPCE].	-

5.2 Reconfigure CPC as SRAM

First, configure the CPC as SRAM in the same manner as described for U-boot above.

However, when performing these steps during Linux kernel initialization the kernel halts after `dcbf` operations. This behavior remains to be analyzed and debugged.

6 Revision history

This table summarizes revisions to this document.

Table 3. Revision history

Revision	Date	Description
0	07/2013	Initial public release.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number AN4749
Revision 0, 07/2013