AN12628

Optimizing Memory Copy Routines

Rev. 0 — 11/2019 Application Note

Contents

1 Introduction

Many applications frequently copy substantial amounts of data from one area of memory to another, using the memcpy() C library function. For example, in the data communication field, communication protocols are divided in many layers. And the payload in the package needs to be fragmented or reassembled between different layers as per the protocol specifications. Due to the complexity of multi-core communication system, it's very difficult to use zero-copy to exchange data among all layers. As this can be quite time consuming, the performance of memory copy routines might be very important in modern communication software. In multi-media application, massive video and audio data needs to be move from one IP module (GPU SRAM) to main DDR memory or vice-versa, even with format conversion.

It may be worth spending some time optimizing the functions that do this. There is no single 'best method' for implementing a copy routine, as the performance depends on many factors. This application note explains these factors.

The concept is suitable for other similar algorithms as well, such as CRC calculate for payload.

Please note that all the performance data in this document is very related to the dedicated platform and configurations. All the data referred here is for comparing only, please take it as it is.

2 Hardware principle of memory copy

The algorithm of memory copy is very straight forward. According to the computer principle, the execution unit of the CPU loads the content from source memory to internal registers. The content is loaded from outside the CPU. For modern computer, latency is high and requires hundreds of CPU cycle for one loading action. Then, the register content is stored to target memory, which is also outside of CPU.

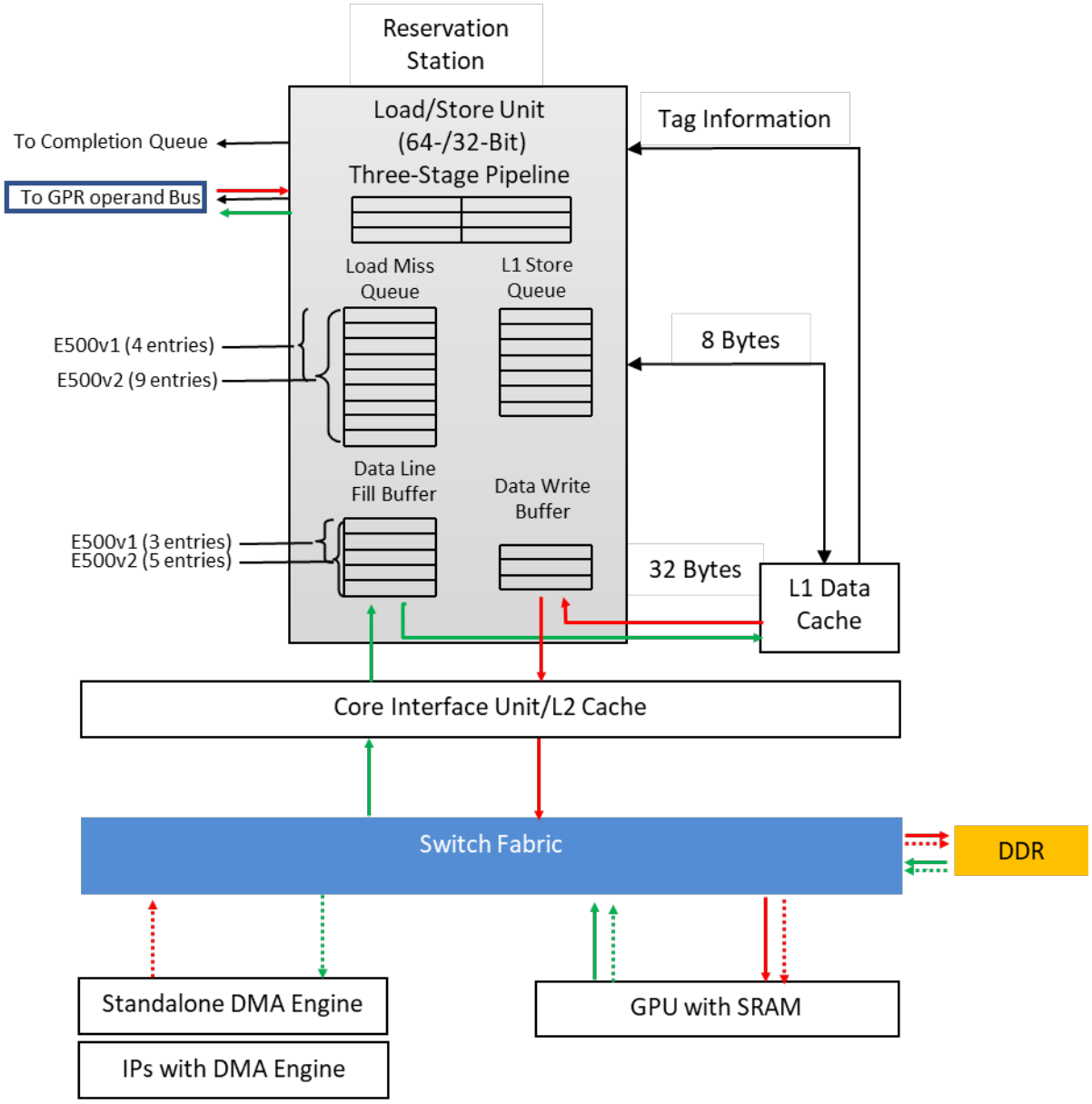

Figure 1. Common computer architecture for memory copy

2.1 Instruction pipeline

The super scalar processor can issue two or more instructions and complete multi-instruction per clock cycle. Instructions complete in order but can execute out of order. In order to parallelly execute instructions, you need to consider the number of execute units, depth of instruction issue queue, complete queue, register dependency, instruction latency. For example, Figure 2 shows the PowerPC ISA instruction pipeline and Figure 3 shows Arm® architecture individually.

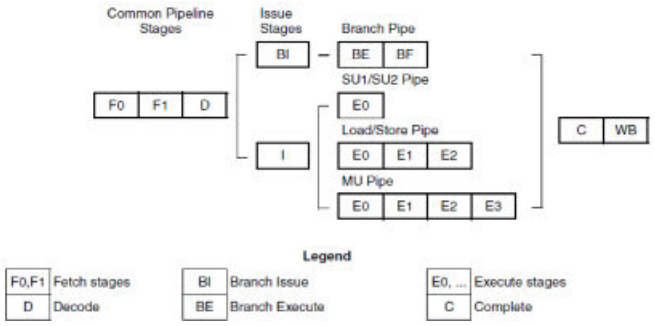

Figure 2. Instruction pipeline diagram of e500 core

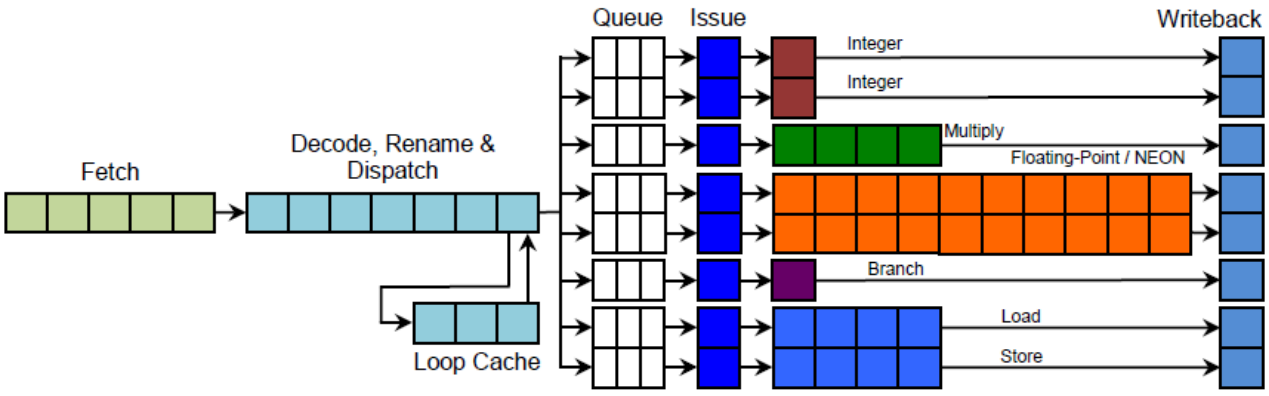

Figure 3. Instruction pipeline diagram of Arm A15 core

2.2 Load store unit

The LSU executes all load and store instructions and provides the data transfer interface between the GPRs, FPRs, and the cache/memory subsystem. The LSU calculates effective addresses, performs data alignment, and provides sequencing for load/store string and multiple instructions. Load and store instructions are issued and executed in program order; however, the memory accesses can occur out of order. Synchronized instructions are provided to enforce strict ordering. Cacheable loads, when free of data bus dependencies, can execute out of order with a maximum throughput of one per cycle and with a two-cycle total latency (e500 core). Data returned from the cache is held in a renamed register until the completion logic commits the value to a GPR or FPR. Stores cannot be executed in a predicted manner and are held in the store queue until the completion logic signals that the store operation is to be completed to memory. The core executes store instructions with a maximum throughput of one per cycle and with a three-cycle total latency (e500 core). The time required to perform the actual load or store depends on whether the operation involves the cache, system memory, or an I/O device.

There are more than one load store execute units plus more floating-point and SIMD units.

2.2.1 Address align

The operand of a single-register memory access instruction has an alignment boundary equal to its length. An operand's address is misaligned if it is not a multiple of its width. Some instructions require their memory operands to have certain alignment. In addition, alignment can affect performance.

Table 1 shows the address align in PowerPC architecture.

Table 1. Address characteristics of aligned operands

Operand	Length	Least 4-bit	Support load/store instruction		
			Classic[1]	64-bit vector[2]	64-bit float point[3]
Byte	1 byte	0bxxx1	Uncertain[4]	Exception	Exception
Half Word	2 bytes	0bxx10	Uncertain	Exception	Exception
Word	4 bytes	0bx100	Good	Exception	Degradation[5]
Double Word	8 bytes	0bx000	Good	Good	Good

1. The classic load/store instructions include, not limited to, lbz/lhz/lwz/stb/sth/stw/ld/std.
2. The 64-bit vector load/store instructions like evldd/evstdd.
3. The 64-bit float point load/store instructions like lfd/stfd
4. If the access crosses the double word boundary, there is performance degradation, otherwise, the performance is good.
5. There is at least two cycles penalty for one instruction.

The Armv8-A architecture allows many types of load and store accesses to be arbitrarily aligned. The Arm® Cortex®-A57 processor handles most unaligned accesses without performance penalties. However, there are cases which reduce bandwidth or incur additional latency, as described below.

- Load operations that cross a cache-line (64-byte) boundary
- Store operations that cross a 16-byte boundary
- Unaligned load/stores cost one more cycle in NEON (64 for 1/3 register, 128 for 2/4 register operations)

2.2.2 Address collision

In Power ISA, the instruction in EX1 stage is a load and has a partial or full address collision with an access in the store queue (a load-on-store collision), which may cause an LSU instruction to replay and at least 3 cycles bubbles in LSU.

The partial address collision means the least significant 12 bits of the load instruction is overlapping with one address in the store queue and the most significant 24 bits are different. If the most significant 24 bits are same, it means full address collision and more cycles of penalty.

2.2.3 SIMD vector engine

In e500 core, the Signal Processing Engine (SPE) is one of Auxiliary Processing Units (APUs), which is a 64-bit, two-element, single-instruction multiple-data (SIMD) ISA. And the 32 general purposed registers in e500 core are all 64-bit, which are accessed by SPE APU. So, this IP block can be used to load/store 64-bit data in memory copy routine. There is one limitation for this kind of vector load/store instruction. The accessing address must be double word (64-bit) aligned. Please refer to Table 1. Besides the align limitation, there are other limitations as well, such as execution unit and register allocation.

Armv7 architecture introduced the Advanced SIMD extension as an optional extension to the Armv7-A and Armv7-R profiles. It extends the SIMD concept by defining groups of instructions operating on vectors stored in 64-bit D, doubleword, registers and 128-bit Q, quadword, vector registers.

The implementation of the Advanced SIMD extension used in Arm® processors is called NEON, and this is the common terminology used outside architecture specifications. NEON technology is implemented on all current Arm® Cortex®-A series processors.

2.3 Data cache control

The load and store instructions can operate with single cycle throughput when each load and store occur to a cache line that is already established in the data cache. To improve the efficiency, there is a cache layer in CPU. Then, the unit of read and write external memory of CPU is burst to one cache-line, for example 64 bytes, instead of one 32-bit or 64-bit word.

It's good to improve the read performance, but it's not always good for write. In order to guarantee the right of write, the destination content in DDR should be read to cache before the write of load-store execute unit.

While the computer executes the memory copy routine, for the first write action of one cache line of the destination address, the hardware loads the old content of the destination address from the external memory, such as DDR into cache due to the cache miss. If the whole cache line of this destination address is over written by the memory copy routine, the cache line loading is an unnecessary action.

2.3.1 Software cache pre-fetch

The CPU cores provide a few cache management instructions that cause the hardware to establish cache lines ahead of the loads and stores.

In Armv8-A architecture, it is also possible for the programmer to give an indication to the core about which data is used in the future. The Armv8-A provides preload hint instructions, like "*PRFM <prfop>, addr*".

- The PRFM instruction retires when its linefill is started, rather than waiting for the linefill to complete. This enables other instructions to execute while the linefill continues in the background.
- For normal memory, up to six 64-byte cache line requests can be outstanding at a time. While those requests are waiting for memory, loads to different cache lines can hit the cache and return their data.

Similarly, in PowerPC ISA, the cache management instructions act as issue and retire also, without the need to wait for the linefill to complete.

2.3.1.1 Cache pre-fetch for load

The Data Cache Block Touch (dcbt) instruction in Power ISA provides a hint to the CPU that a program is likely execute a load instruction from an address. For a memory copy routine, the dcbt instruction can be used to preload source data from memory into the data cache so that subsequent load instructions result in cache hits and execute quickly.

The Armv8-A ISA provides the "PRFM PLD" instruction, to signal to the memory system that memory load from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory access when the real load does occur.

2.3.1.2 Cache pre-fetch for store

The Data Cache Block Set to Zero (dcbz) instruction in Power ISA establishes a cache block in the data cache and fills it with zero bytes without accessing memory if the effective address of the dcbz is marked cacheable and non-write-through. This instruction is often used to efficiently zero large sections of memory without first fetching that memory into the cache. In a memory copy routine, the dcbz instruction is used to simply establish a destination data block in the cache so that subsequent store instructions result in cache hits. In addition, unnecessary loads from the destination memory are eliminated.

The Armv8-A architecture introduces a Data Cache Zero by Virtual Address (DC ZVA) instruction. This enables a block of 64 bytes in memory, aligned to 64 bytes in size, to be set to zero. If the DC ZVA instruction misses in the cache, it clears main memory, without causing an L1 or L2 cache allocation. So, it's not adapted to the memory copy scenario.

Secondly, the Power ISA provides the Data Cache Block Touch for Store (dcbtst) instruction that behaves the same as dcbt, except the hint assumes that software will soon write to the block. Responsibly, the Armv8-A architecture provides the "PRFM PST" instruction to pre-fetch memory for store in the future as well.

2.3.2 Hardware cache pre-fetch

The Load/store unit includes a hardware prefetcher that is responsible for generating prefetches targeting both the L1D cache and L2 cache.

The load side prefetcher uses a hybrid mechanism which is based on both physical-address (PA) and virtual-address (VA) prefetching to either or both of the L1D cache and L2 cache, depending on the memory access patterns.

Prefetching on store accesses is managed by a PA based prefetcher and only prefetches to the L2 cache. The Bus Interface Unit (BIU) in Armv8-A architecture includes logic to detect when a full cache line has been written by the processor before the linefill has completed. If this situation is detected on three consecutive linefills, it switches into read allocate mode. When in read allocate mode, loads behave as normal and can still cause linefills, and writes still lookup in the cache but, if they miss, they write out to L2 rather than starting a linefill. The BIU continues in read allocate mode until it detects either a cacheable write burst to L2 that is not a full cache line, or there is a load to the same line as is currently being written to L2. So, it's very helpful to skip unnecessary destination memory pre-load in memory copy scenario while size is more than three consecutive cache lines.

Unfortunately, there is no hardware cache pre-fetch in PowerPC core based SoC devices from NXP.

2.4 DMA accelerator engine

In the SoC devices, there are some kinds of DMA hardware engines, which can act as bus master to move data from one bus slave to another slave. The advantages of DMA copy are:

- Offload the work from core, and avoid populating the cache of core and platform.
- The bandwidth of DMA to/from main memory is bigger than the core bus interface because of the bigger burst size, for example 256 bytes instead of 64-byte cache linefill from core.
- The software is not limited to the address align and more user friendly, although aligned copy has better performance. In DMA engine, the stride copy is a very common feature and is very helpful for end users as well.

The disadvantages of DMA copy are:

- The address must be physical address instead of virtual address from user space application. The user must keep it in mind.
- The DMA hardware must be managed by the kernel device driver, the context switch between user space and kernel space will have some cycles to cost. So, it's not suitable for small block size copy.

The qDMA controller in LX2160A transfers blocks of data between one source and one or more destinations. The blocks of data transferred can be represented in memory as contiguous or non-contiguous using scatter/gather table(s). Channel virtualization is supported through enqueuing of DMA jobs to, or dequeuing DMA jobs from, different work queues.

2.4.1 Stride mode in qMDA

One of the special benefits of DMA copy is the stride mode support, as shown in *Figure 4*.

```
int xres = xxx; //分辨率x轴
int yres = yyy; //分辨率y轴
int stride = 0, stride_align = 64;
// Caculate stride
do {
    stride += stride_align;
}while(stride < xres);

for(int i = 0; i < yres; ++ i )
{
    memcpy((char*)dst + i * xres * 4, src + i * stride * 4, xres * 4 );
}
```

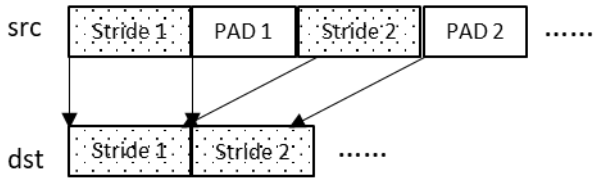

Figure 4. qMDA stride mode

2.4.2 Device driver for qDMA

Based on standard qDMA driver for DPAA2.0 framework on LX2160A device from NXP, we enhance and implement an additional system call to do memory copy for user space applications. We use multi-queue to support channel virtualization. Multi-thread applications on multi-core can call this service freely without resource spin lock and share the qDMA hardware bandwidth transparently, refer to Figure 5.

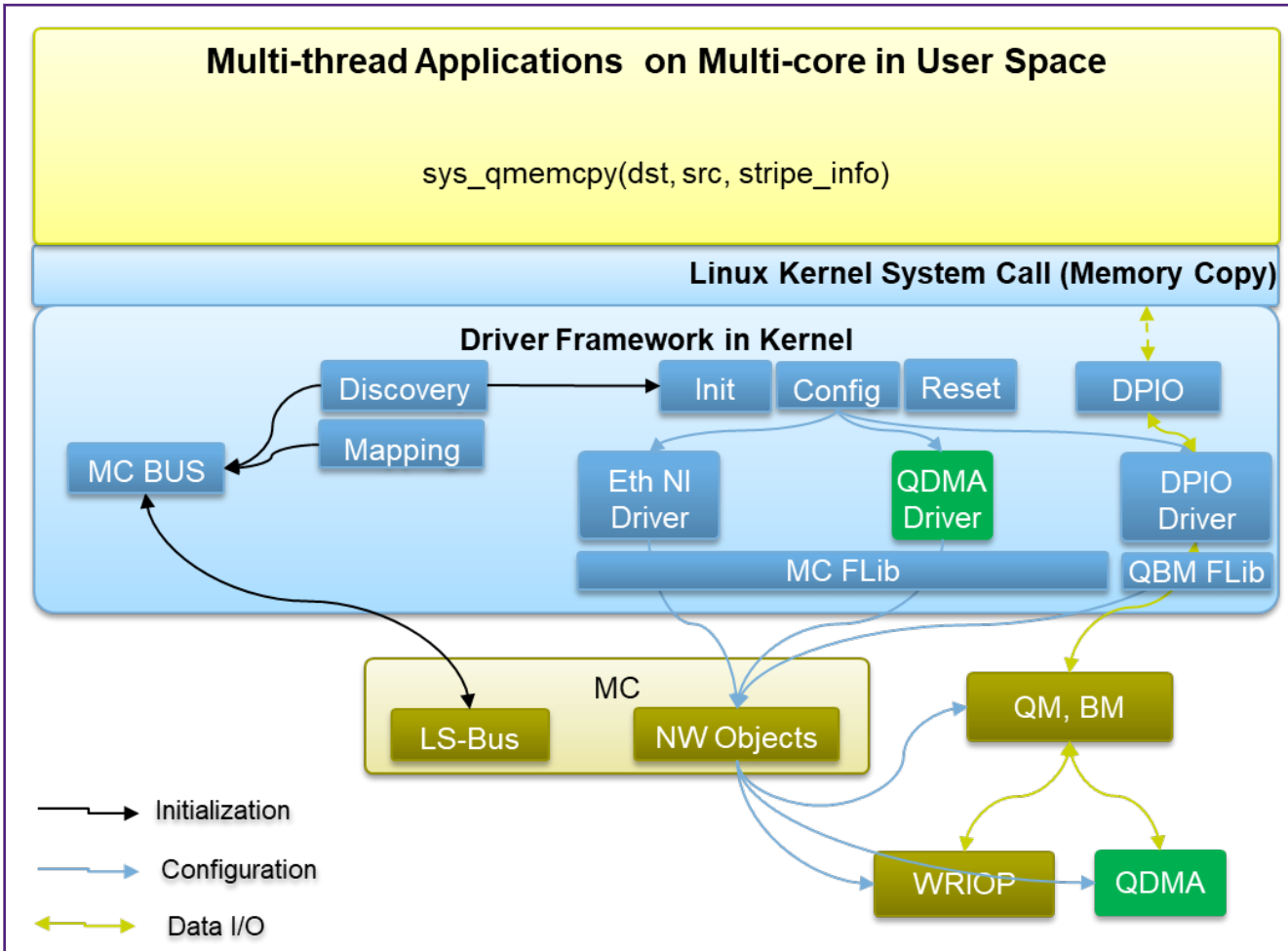

Figure 5. QDMA driver on LX2160

2.4.3 qDMA data path

Because there are special data paths from qDMA to PCIe interface, the memory moves between DDR and PCIe interface benefit.

If the qDMA moves data from DDR to DDR, the action consumes double bandwidth of DDR controller and main bus, one for read and one for write. While the qDMA moves data from DDR to PCIe, there is only one DDR read on DDR controller and one bandwidth on main bus, then data moves to PCIe controller directly from qDMA to PCIe controller, because there is a special bus between PCIe and qDMA.

Refer to Figure 6 and Figure 7. It's similar to legacy DMA in PowerPC based SoC.

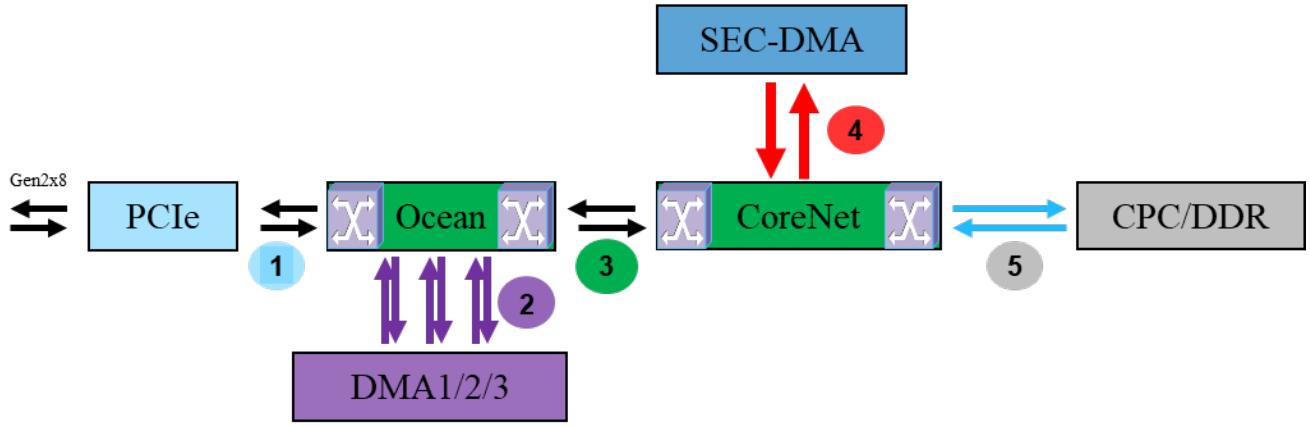

Figure 6. Legacy DMA engine in T4240

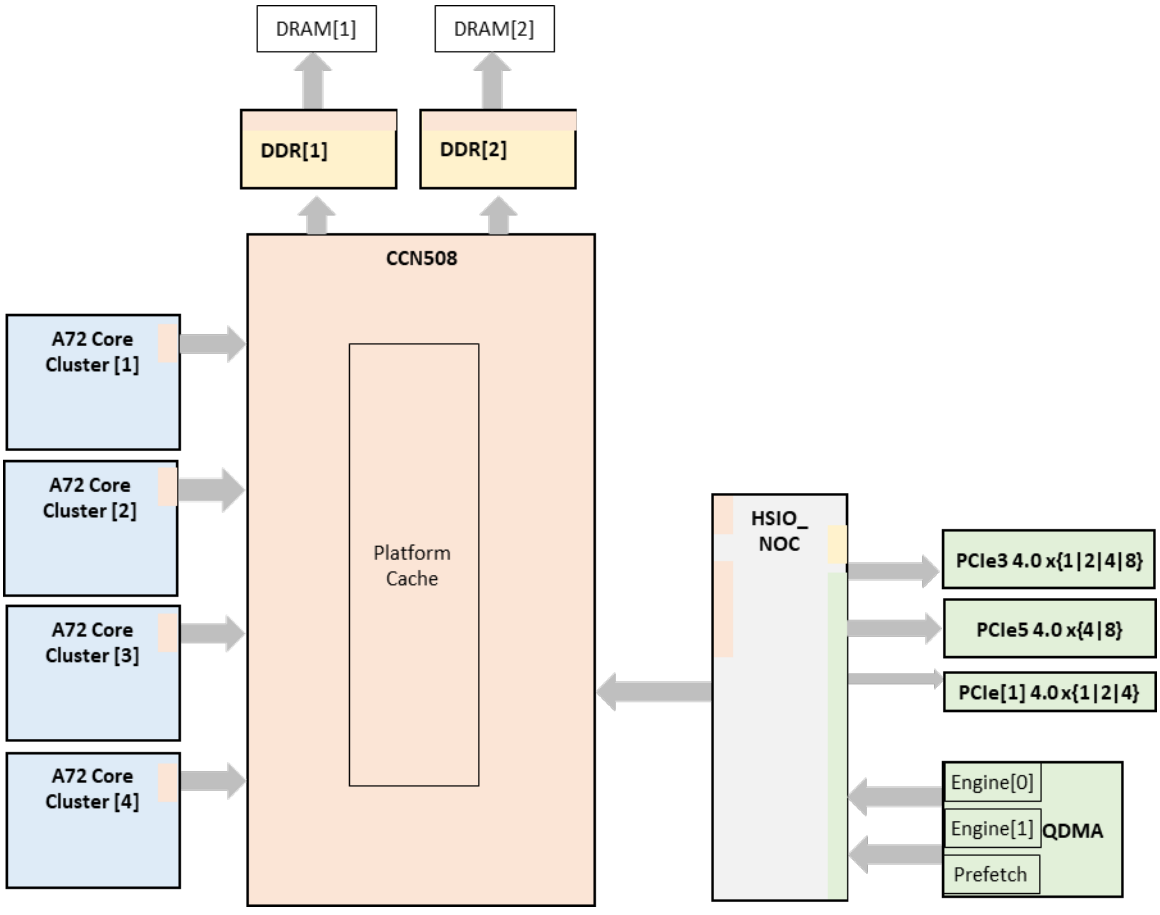

Figure 7. qDMA engine in LX2160

2.5 Other tunings

At same time, modern CPU uses the TLB to implement MMU feature. The CPUs use hardware TLB walk to improve the performance, but still some ISAs use software TLB walk. This document does not explain about the penalty of TLB miss here. It is assumed TLB is hit in all scenarios.

This document does not explain the topic of optimizing on DDR controller, system bus, nor the application itself. Although it's obvious these ways absolutely could improve the performance of memory copy routines.

Also, it is not considered how to reduce the L2/L3 cache pollution which is caused by the memory copy routines to improve the performance of whole system.

3 Optimizations for small size block copy

For small size block copy, to say below three or four cache-lines, the hardware cache pre-fetch does not work well. So, it's very important to use software cache pre-fetch to stream the multi cache reading, and cache zero to setup the target address in data cache to avoid unnecessary dirty read. Because the cache prefetch instructions only need one cycle to trigger the hardware to start the cache line fill, and multiple outstanding requests to be serviced by the DDR controller in parallel.

At same time, take care about the instruction efficiency and schedule will obviously improve the performance, for example, execute unit dependency, register dependency, address collision and align, etc. The SIMD instruction will reduce the number of instructions to benefit the memory copy routines.

Here is a typical example on PowerPC e500. So, with the code in Table 2, we get the performance data 16 cycles per 32-byte (without cache prefetch instruction), when source address is half-word-aligned (for example, like 0x2002), and cache is hit. In comparison to copying byte-by-byte, it takes about 96 cycles per 32-byte on MPC8572. Please note, several lines of code need to be added before and after this loop to make the function to be right, which is not shown in this table. The Figure 8 explains what the code in Table 2 is doing, to copy four 8-bytes in one loop.

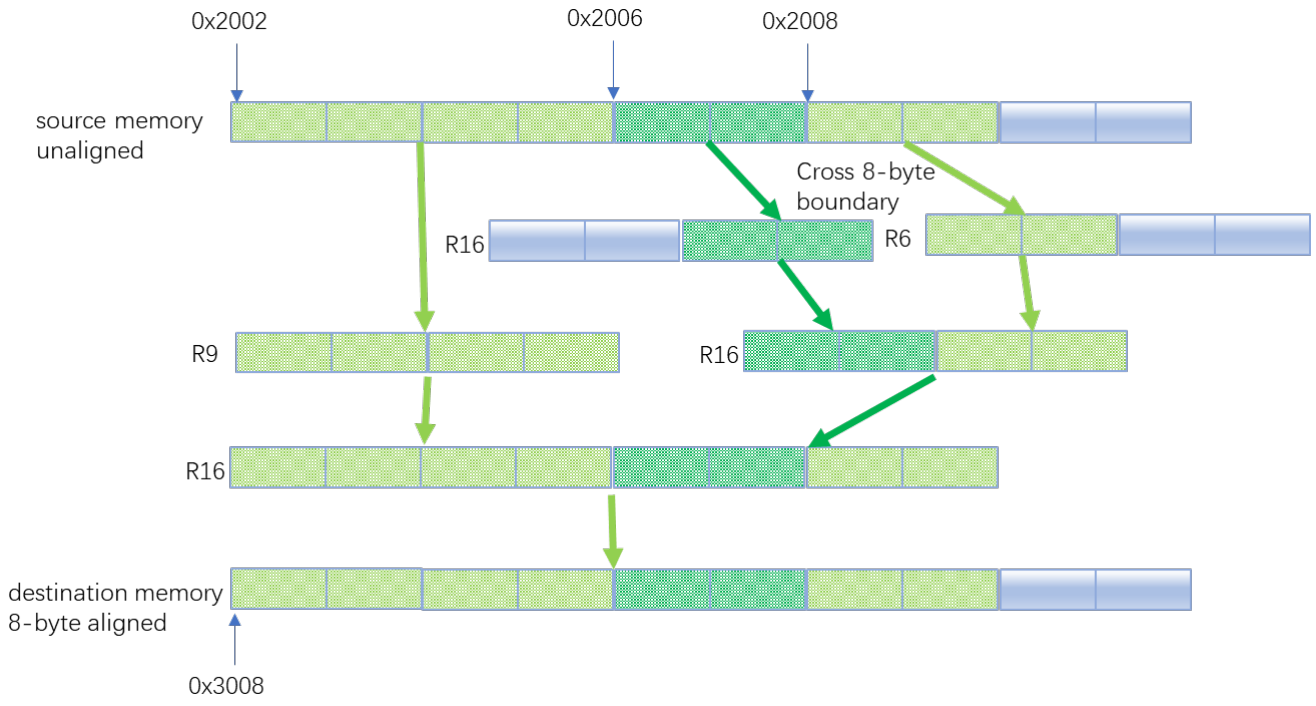

Figure 8. Memory copy on e500 with SPE

Table 2. Use 64-bit SPE vector instruction on 32-bit e500

line Number	Label	Instruction	Comments
0		addi r6,r6,-32	

Table continues on the next page...

Table 2. Use 64-bit SPE vector instruction on 32-bit e500 (continued)

line Number	Label	Instruction	Comments
1		dcbt 0, r12, r4	/* 1 cycles, might multi cache lines */
2		dcbz r12,r6	/* 2 cycles, might multi cache lines */
3	loop_32byte_spe:	lhz r0,4(r4)	cycle #1
4		rlwimi r7,r16,16,0,15	/* at least 2 cycles after r7/r16, lines #24, #25 */
5		lhz r16,6(r4)	cycle #2
6		addi r6,r6,32	
7		lwz r9,0(r4)	cycle #3
8		addic. r10,r10,-1	
9		evmergelo r8,r8,r14	/* after 5 cycles of r14/r8, lines #26, #20 */
10		evstdd r8,0(r6)	cycle #4
11		lhz r8,12(r4)	cycle #5
12		lhz r11,14(r4)	cycle #6
13		rlwimi r16,r0,16,0,15	/* at least 2 cycles after r16/r0, lines #24, #3 */
14		evmergelo r15,r15,r7	/* after 5 cycles of r7/r15, lines #3, #29*/
15		evstdd r15,8(r6)	cycle #7
16		lwz r15,8(r4)	cycle #8
17		lhz r0,20(r4)	cycle #9
18		rlwimi r11,r8,16,0,15	/* at least 2 cycles after r8/r11, lines #11, #12*/
19		lhz r14,22(r4)	cycle #10
20		lwz r8,16(r4)	cycle #11
21		addi r4,r4,32	
22		evmergelo r9,r9,r16	/* after 5 cycles of r9/r16, lines #6, #13*/
23		evstdd r9,16(r6)	cycle #12
24		lhz r16,-4(r4)	cycle #13
25		lhz r7,-2(r4)	cycle #14

Table continues on the next page...

Table 2. Use 64-bit SPE vector instruction on 32-bit e500 (continued)

line Number	Label	Instruction	Comments
26		rlwimi r14,r0,16,0,15	/* at least 2 cycles after r0/r14, lines #17, #19*/
27		evmergelo r15,r15,r11	/* after 5 cycles of r11/r15, lines #16, #18*/
28		evstdd r15,24(r6)	cycle #15
29		lwz r15,-8(r4)	cycle #16
30		bdnz loop_32byte_spe	
		addi r6,r6,32	

For Armv8-A, it's straight forward, because it's 64-bit and without aligned limitation. The NEON instructions are showed below for your reference.

Table 3. 64-bit Armv8 w/o 128-bit NEON involved

line Number	Label	Instruction	Comments
0		prfm pldl1strm, [x1,#144]	/* 1 cycles, might multi cache lines */
1		prfm pstl1strm, [x0,#144]	/* 1 cycles, might multi cache lines */
2		sub x1, x1, #0x10	
3		sub x0, x0, #0x10	
4	loop_64byte:	ldp x10, x11, [x1,#16]	ld4 {v0.16b-v3.16b}, [x1], #64
5		ldp x8, x9, [x1,#32]	
6		ldp x6, x7, [x1,#48]	
7		ldp x4, x5, [x1,#64]	
8		stp x10, x11, [x0,#16]	st4 {v0.16b-v3.16b}, [x0], #64
9		stp x8, x9, [x0,#32]	
10		stp x6, x7, [x0,#48]	
11		stp x4, x5, [x0,#64]	
12		add x3, x3, #0x80	

Table continues on the next page...

Table 3. 64-bit Armv8 w/o 128-bit NEON involved (continued)

line Number	Label	Instruction	Comments
13		ldp x10, x11, [x1,#80]	ld4 {v0.16b-v3.16b}, [x1], #64
14		ldp x8, x9, [x1,#96]	
15		ldp x6, x7, [x1,#112]	
16		ldp x4, x5, [x1,#128]!	
17		stp x10, x11, [x0,#80]	st4 {v0.16b-v3.16b}, [x0], #64
18		stp x8, x9, [x0,#96]	
19		stp x6, x7, [x0,#112]	
20		stp x4, x5, [x0,#128]!	
21		cmp x12, x3	
22		b.cs loop_64byte	

You need to arrange instructions before and after the main loop, to make sure the destination address is appropriately aligned, for example cache-line aligned, and try your best to reduce the number of instructions and aligned access penalty.

4 Optimizations for big size block copy

Armv8-A provides load and store instructions to avoid cache population. But the real requirements depend on the customer's application. For example, while the memory copy routine is invoked, the source address or destination address will be accessed in the near future or not.

Because Armv8-A has built-in cache pre-fetch engine in the core, you do not need to use cache pre-fetch instructions in the memory copy routine while the copy size is more than three or four cache lines if you do not consider the application requirements after the memory copy routine is evoked.

For PowerPC architecture, you should use DCBT instruction to do cache pre-fetch ahead of two or three cache lines for real load operation and use DCBZ instruction to do destination cache setup to avoid the cache miss before real store operation. It's obvious to improve the performance. Table 4 shows performance on PowerPC. According to this data, the DMA performance is much better.

Table 4. Memory copy improvement for cache-missed[1]

Core Cycles Number		DMA Copy 4M Block	Typical Byte Copy	64-bit SPE Copy	64-bit SPE Copy with prefetch
One DDR One Logical Bank	8-byte aligned	48	350	350	140
	4-byte aligned	59	350	350	140
	2-byte aligned	58	350	350	140

Table continues on the next page...

Table 4. Memory copy improvement for cache-missed[1] (continued)

Core Cycles Number		DMA Copy 4M Block	Typical Byte Copy	64-bit SPE Copy	64-bit SPE Copy with prefetch
One DDR Two Logical Banks	8-byte aligned	36	300	230	70
	4-byte aligned	42	300	230	70
	2-byte aligned	42	300	230	70
Two DDRs Two Logical Banks	8-byte aligned	33	190	140	35
	4-byte aligned	38	190	140	35
	2-byte aligned	38	190	140	35

1. It's the core cycles to do 32-byte copy, both source and destination address are cache-missed. This MPC8572 board settings are: core/ccb/ddr frequency equals to 1250 MHz/500 MHz/266 MHz, 256M bytes per DDR controller, 13 row, logical banks, 10 cols, and 3-bit for byte selects (64-bit bus), DDR2 devices, CASLAT: 11 clocks, WR_LAT: 5 clocks, ACTTORW: 6 clocks.

For Armv8-A ISA, the performance data of QDMA on LX2160 is not competitive with LIBC memory copy due to one QDMA performance bug in Rev. 1.0 silicon. It is expected to have a good offloading performance with QDMA on Rev. 2.0 silicon after this hardware bug is fixed.

5 Benchmark bed

5.1 Instruction latency and throughput

The performance monitor counter is used to inspect the instruction execution. Figure 7 shows the sample code for your reference.

```
uint64_t get_cycles(void)
{
    uint64_t value;
    asm volatile("mrs %0, pmccntr_el0" : "=r" (value));
    return value;
}

uint32_t inline get_instructions(void)
{
    uint32_t value;
    asm volatile("msr pmselr_el0, %0" :: "r" (0));
    asm volatile("isb\n");
    asm volatile("mrs %0, pmxevcntr_el0" : "=r" (value));
    return value;
}
throughput()
{
    curr = getTimeStamp();
    inst1 = get_instructions();
    cycle1 = get_cycles();

    memcpy(dst,src,copy_size);

    cycle2 = get_cycles() - cycle1;
    inst2 = get_instructions() - inst1;
    curr = getTimeStamp() - curr;
}

```

Figure 9. Performance monitor counter on Armv8

5.2 Throughput of memory copy routine

In order to benchmark the throughput of memory copy routine, setup a multi-task framework, to synchronize the execution of the multi-task on multi-cores, allocation memory for every one of the tasks, verify the function of memory copy, collect and output the performance of function evokes.

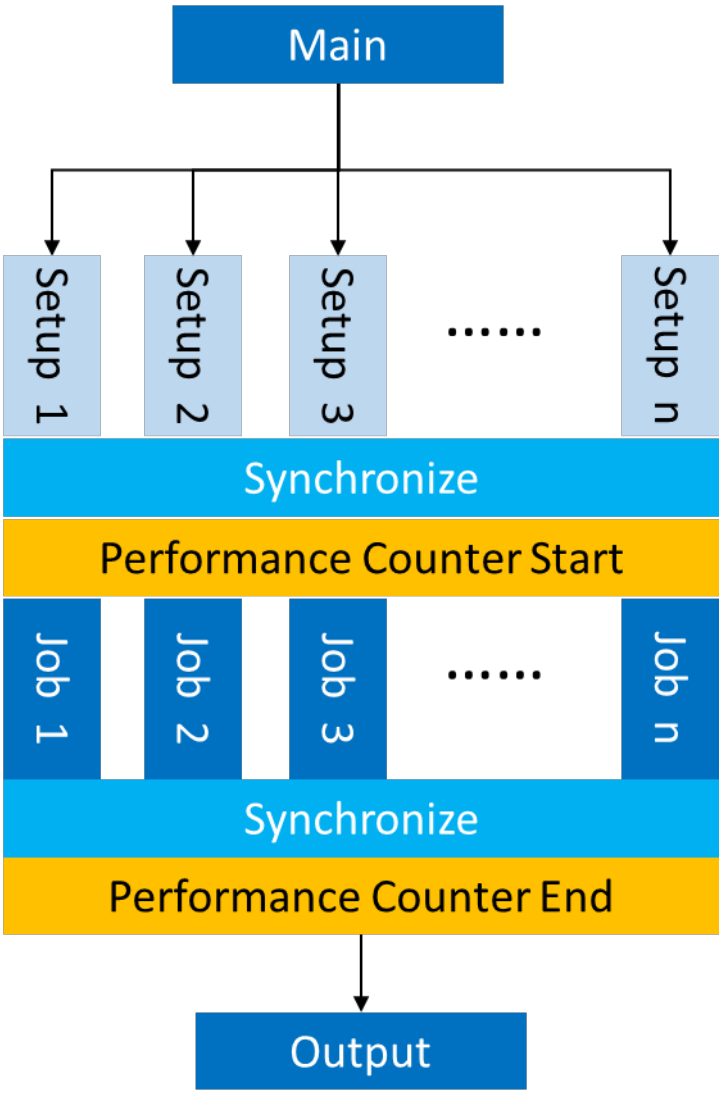

Figure 10. Throughput benchmark framework

6 Reference

1. AN2665: e500 Software Optimization Guide (eSOG), Rev. 0, 04/2005
2. PowerPC e500 Core Family Reference Manual, E500CORERM, Rev. 1, 4/2005
3. E500mc Core Reference Manual, Rev F, 01/2010
4. EREF: A Programmer's Reference Manual for Freescale Embedded Processors, EREFRM, Rev. 1, 12/2007
5. Signal Processing Engine (SPE), Programming Environments Manual, SPEPEM, Rev. 0, 01/2008
6. SYSTEM V APPLICATION BINARY INTERFACE, PowerPC Processor Supplement, Revision A, September 1995
7. Coretex-A57 Software Optimization Guide, January 28, 2016
8. NEON Programmer's Guide, Version: 1.0
9. What is the fastest way to copy memory on a Cortex-A8?
10. Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile

11. Arm® Cortex®-A72 MPCore Processor, Revision: r0p3, Technical Reference Manual

7 Revision History

Table 5 provides a revision history for this application note.

Table 5. Document Revision History

Revision	Date	Topic cross-reference	Change desription
Rev 0	11/2019		Initial release

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 11/2019
Document identifier: AN12628