Intel® IXP42X Product Line and IXC1100 Control Plane Processor: Memory Management Unit and Cache Operation

Application Note

July 2004

Order Number: 252676-002

Contents

Figures

Tables

Revision History

Date	Revision	Description
July 2004	002	Updated Intel® product branding.
June 2003	001	Initial release of this document.

1.0 Introduction

The Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor incorporate silicon blocks that enhance computing performance. Those blocks include:

- Cache and Memory Management Unit (MMU)
- Three dedicated network-processing engines
- AHB Queue Manager

This document discusses the software implications of the Intel® IXP42X product line and IXC1100 control plane processors' Memory Management Unit and cache capabilities. It also covers the risks associated with porting Intel® IXP22x software components that were not designed with caching in mind.

When discussing the Intel® IXP42X product line and IXC1100 control plane processors' board-support package (BSP) issues — such as memory maps and caching policies — this document will refer to the Intel® IXDP425 Network Processor Development Platform.

1.1 Scope

This document's scope is as stated in the preceding "Overview" section. While the Intel XScale® Core memory accesses are affected by the SDRAM controller as well as the MMU, the issues related to the SDRAM controller are considered outside the scope of this document.

The document is intended for system and software engineers who are developing software or board-support packages (BSPs) that improve the performance of integrated access devices such as DSL modems, residential gateways, and set-top boxes.

1.2 Related Documents

Document	Document Number
Intel® IXDP425 / IXCDP1100 Development Platform Quick Start Guide	253177
Intel® IXP4XX Product Line Programmer's Guide (1.1)	252539
Intel® XScale™ Core Developer's Manual	273473
Wind River VxWorks* Network Programmer's Guide*	N/A

1.3 Introduction to Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor

The Intel® IXP42X product line includes multi-functional processors that incorporate many advanced architecture features, including an industry-standard, 32-bit PCI controller, Universal Asynchronous Receiver and Transmitter (UART), PC133 SDRAM memory controller, and interrupt controller. The processors also incorporate the Intel XScale core technology, compliant with the ARM* Version 5TE instruction set architecture (ISA), Universal Serial Bus (USB), and Universal Test and Operation PHY Interface for ATM (UTOPIA) and have General-purpose input/output (GPIO), 133-MHz internal bus, Advanced High-Performance Bus (AHB) bridges, timers, an 8-Kbyte Queue Manager, internal-bus Performance Monitoring Unit (PMU), network processor engines (NPEs), and industry-standard Media Independent Interfaces (MII).

With the highly integrated solution and rich feature set, the Intel® IXP42X product line and IXC1100 control plane processors deliver a high-performance system-on-chip that reduces the cost of system implementations.

The Intel® IXP42X product line and IXC1100 control plane processors can operate at a variety of frequencies, allowing systems designers to trade off performance for power consumption. The Intel XScale core and the internal components are clocked from an internal PLL. The internal and external interfaces use a 66-/133-MHz, 32-bit data bus; a 32-bit address bus; and control signals that optimizes performance between the Intel XScale core and peripheral logic.

1.4 Acronyms

AHB	Advanced High-performance Bus
AP	Access Permission
BSP	Board Support Package
CAM	Contents Addressable Memory
CPU	Central Processing Unit
DMMU	Data Memory Management Unit
IMMU	Instruction Memory Management Unit
ISR	Interrupt Sub-Routine
LAN	Local Area Network
MAC	Multiply/Accumulate
MMU	Memory Management Unit
MSR	Monsoon Software Release
NPE	Network Processing Engine
WAN	Wide Area Network

1.5 Glossary

CAM	Content Accessible Memory. CAM is an aggregation of cells each containing an in-built comparator. A CAM has three modes of operations: read, write, and match. In read or write modes, the CAM behaves like conventional memory, the benefit of CAM is highlighted in match mode. Match mode allows a particular bit pattern and a mask to be submitted. All CAM cells are then searched simultaneously and the cells that match the desired pattern are passed to a validation block. The search speed that CAM provides makes it a vital component for applications that require fast table lookup such as virtual memory translators, caches, and data compression.
Dirty Bits	Each cache line in the data and mini-data cache has two dirty bits associated with it. One dirty bit is associated with the lower 16 bytes of the cache line and the other with the upper 16 bytes. When the write policy — associated with a given cache line — is write-back, the dirty bit is set when any of the 16 bytes associated with it is written to in cache. When the round-robin replacement algorithm determines that a given cache line has to be replaced by a new one in cache, the hardware causes the 16 bits — that have their dirty bit set — to be written back to external memory, preserving memory coherence. Basically, these bits are flags which indicate that the data in data cache or mini-data cache is more up-to-date than what is in external memory.
Fill Buffer	The Intel® IXP42X product line and IXC1100 control plane processors' Intel XScale core contains a four 32-byte-entry fill buffer. The fill buffer stores a cache line, 32 bytes, from external memory every time an Intel XScale core read operation causes a cache miss. Whenever a read operation causes a cache miss, the entire cache line containing the desired word is loaded into one of the fill buffer entries. When the entire cache line is loaded from external memory, the fill buffer entry is written to cache and marked as free for future requests. Up to four entries can be outstanding before the Intel XScale core has to stall and wait for a fill buffer entry. **NOTE:** In order to reduce delay, the critical word is always returned to the Intel XScale core first, see "Read Policy" on page 20.
Privileged Modes	The majority of a program will execute in user mode, the only way to switch to a privileged mode is for an exception to occur. The Intel XScale core supports five privileged modes: fast interrupt, normal interrupt, memory aborts, attempted execution of an undefined instruction, and a software interrupt. In privileged modes, the program has access to more instructions and registers than in user mode.
User Mode	Most programs will be executed in user mode — not within the context of an ISR. Certain instructions and registers are not accessible, when code is executing in user mode.

2.0 Caching and MMU Features

The Intel XScale core of the Intel® IXP42X product line and IXC1100 control plane processors is an ARM* V5TE-compliant microprocessor. The Intel® IXP42X product line and IXC1100 control plane processors' Intel XScale core incorporates several silicon features that help it deliver high performance, this technical note will concentrate on the memory management facilities in Intel® IXP42X product line and IXC1100 control plane processors' Intel XScale core.

2.1 Memory Management Units

The Intel XScale core implements the Memory Management Unit (MMU) architecture specified in the *ARM Architecture Reference Manual*. The MMU provides access protection and virtual-to-physical address translation.

The MMU architecture also specifies the caching policies for the instruction cache and data cache. These policies are specified as page attributes and include:

- Identifying code as cacheable or non-cacheable
- Selecting between the mini-data cache or data cache
- Write-back or write-through data caching
- Enabling data-write allocation policy
- Enabling the write buffer to coalesce stores to external memory

The Intel XScale core contains one Instruction Memory Management Unit (IMMU) and one Data Memory Management Unit (DMMU). From a functionality point of view, the two MMUs are similar — the only difference being that the former is activated with op-code fetches and the latter activated with memory load/store operations.

The Intel XScale core uses both an instruction Translation Look-Aside Buffer (TLB) and a data TLB to cache the latest translations, accelerating virtual-to-physical address translation.

2.1.1 Virtual to Physical Address Mapping

The Intel XScale core MMU units allows control of memory systems through the TLBs. Entries in these TLBs define the properties of memories spaces.

One of these properties is virtual-to-physical address mapping or translation. A memory address generated in the Intel XScale core is a virtual or logical memory address. One of the roles of the MMU is to map or translate that virtual memory address to an external physical memory address. The physical address is the actual memory location on the address bus being accessed.

Figure 1. Virtual-to-Physical Address Translation

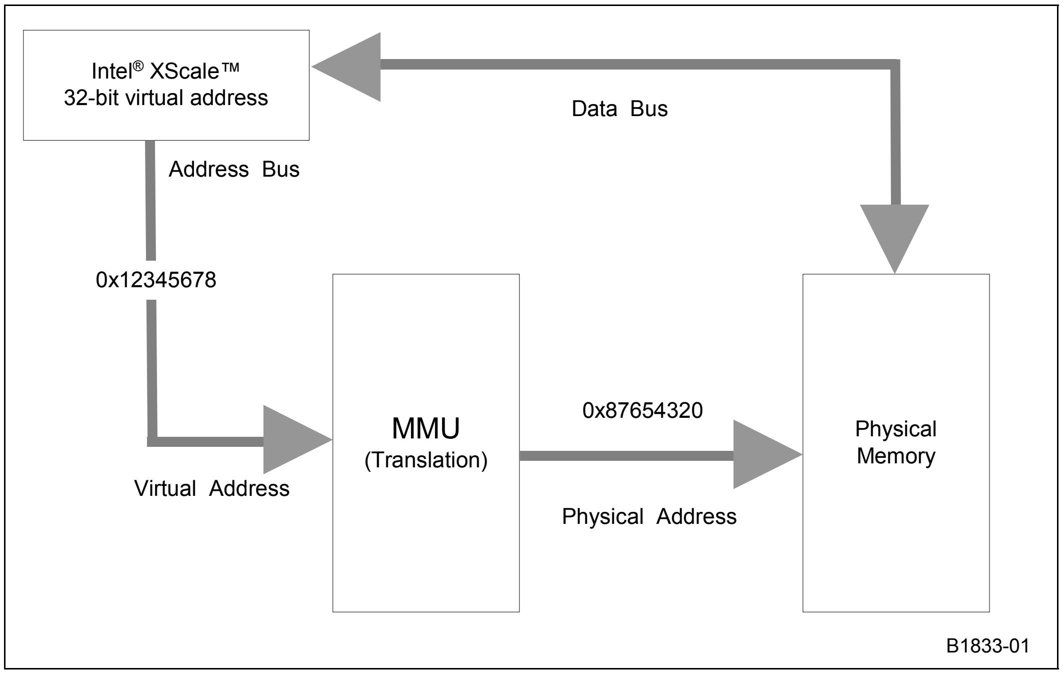


Figure 1 illustrates an example of the translation process:

1. The Intel XScale core puts logical address 0x12345678 on the bus in order to access data or op-code
2. The MMU intercepts the address and performs certain checks on it ("Memory Access Permissions" on page 11).
3. The MMU translates the virtual address to the physical address 0x87654320, to be accessed.

Virtual-to-physical address translation can be useful when the programmer needs to allocate addresses to different processes with potentially conflicting address spaces, or when the programmer wants to use a contiguous address space for an application that uses fragmented memory segments.

Figure 2. Virtual-to-Physical Address Mapping On Intel® IXDP425 Network Processor Development Platform

	Virtual Address Map	Physical Address Map
0xFFFFFFFF		
	Undefined — 800 Mbyte	Undefined — 800 Mbyte
0xCC001000		
	SDRAM Reg — 4 Kbyte	SDRAM Reg — 4 Kbyte
0xCC000000		
	Undefined — 64 Mbyte	Undefined — 64 Mbyte
0C8010x000		
	Peripheral Reg — 64 Kbyte	Peripheral Reg — 64 Kbyte
0xC8000000		
	Undefined — 64 Kbyte	Undefined — 64 Kbyte
0xC4001000		
	Expansion Bus Reg — 4 Kbyte	Expansion Bus Reg — 4 Kbyte
0xC4000000		
	Undefined — 64 Kbyte	Undefined — 64 Kbyte
0xC0001000		
	PCI Reg — 4 Kbyte	PCI Reg — 4 Kbyte
0xC0000000		
	Undefined — 3 Gbyte	Undefined — 3 Gbyte
0x80007000		
	Fast DRAM — 28 Kbyte	Fast DRAM — 28 Kbyte
0x80000000		
	Undefined — 5 Mbyte	Undefined — 5 Mbyte
0x60004000		
	Q Mgr. Registers — 16 Kbyte	Q Mgr. Registers — 16 Kbyte
0x60000000		
	Undefined — 128 Mbyte	Undefined — 128 Mbyte
0x58000000		
	Expansion Bus Reg — 128 Mbyte	Expansion Bus Reg — 128 Mbyte
0x50000000		
	PCI Memory — 128 Mbyte	PCI Memory — 128 Mbyte
0x48000000		
	Undefined — 1 Gbyte	Undefined — 1 Gbyte
0x08000000		
	SDRAM — 128 Mbyte	SDRAM — 128 Mbyte
0x00000000		

For the IXDP425 platform, the BSP instructs the MMU to implement a one-to-one mapping between the virtual and physical address, meaning that if the Intel XScale core accesses virtual address 0x12345678, the physical address 0x12345678 will be accessed.

Figure 2 illustrates the one-to-one mapping between the virtual and physical addresses in the current BSP version. In regards to the figure, be advised that:

- The IXDP425 platform BSP does not define the 28-Kbyte Fast DRAM located in memory space 0x80000000 – 0x80007000, this issue is explained further in "Cache-Line Replacement Policy" on page 17.
- At boot time, the expansion bus and the flash banks that hang off it are located in the memory space 0x00000000 – 0x07FFFFFF.

  During boot operation, the expansion bus and thus flash is moved to address space 0x50000000 – 0x57FFFFFF by writing to the expansion bus control registers.

The virtual to physical address translation scheme is set in the BSP file sysLib.c. (The version of BSP used to get the mapping, illustrated in Figure 2, is TAG_IXP425_CSR_1_0_SQA1_2.)

Once the MMUs have been set up, the translation process is handled entirely by the MMU. The programmer need not be aware of it for any computation that involves the Intel XScale core alone. (The NPEs' memory accesses are not under MMU control and thus only access physical memory, see sections "Intel XScale® Core Submission of Data to the NPE for Transmission" on page 23 and "Cache-Line Replacement Policy" on page 17.)

If a buffer needs to be sent to an NPE, however, for it to be transmitted out to the WAN/LAN, the programmer ought to take special care to convert any pointers inside the mbuf structure — e.g., m_data, m_next as well as the address of the mbuf that is submitted to the Queue Manager — because NPE memory accesses are not seen by the Intel XScale core MMU and thus the translation process will not take effect on NPE memory accesses.

Macros that translate virtual addresses to physical addresses and vice versa are available in the file `xscale_sw/src/include/IxOsCacheMMU.h`:

- IX_MMU_VIRTUAL_TO_PHYSICAL_TRANSLATION(addr)
- IX_MMU_PHYSICAL_TO_VIRTUAL_TRANSLATION(addr)

2.1.2 Memory Access Permissions

The MMU of the Intel® IXP42X product line and IXC1100 control plane processors support memory accesses through sections and pages. Sections and pages are blocks of memory. They differ in sizes.

Table 1. Intel XScale® Core MMU Memory Segments Organization

Name	Size
Section	1 Mbyte
Large Page	64 Kbyte
Small Page	4 Kbyte
Tiny Page	1 Kbyte

The Intel XScale core MMU handles four types of memory segments listed in Table 1. The MMU gives the programmer the ability to define the access permissions for any segment of virtual memory.

From a hardware point of view, each memory section/page has two access permission AP bits associated with it. In conjunction with the ROM R-bit and System S-bit of register 1 of the System Control Coprocessor (CP15), a memory section/page can have the access permissions shown in Table 2.

Table 2. MMU Access Permissions

AP-bits	S-bit	R-bit	Privileged Permissions	User Permissions
00	0	0	No access	No access
00	1	0	Read only	No access
00	0	1	Read only	Read only
00	1	1	Unpredictable†	Unpredictable
01	X	X	Read/Write	No access
10	X	X	Read/Write	Read only
11	X	X	Read/Write	Read/Write

† This AP-bits, S-bit, and R-bit should not be used and if used, theIntel XScale core core behavior will become indeterministic.

From a software perspective, the IXDP425 platform BSP uses VxWorks* `vmLib` library to define memory-management policy. The `vmLib` library does not support all the access level permissions allowed by the MMU. Instead, the library allows the user to define permissions as follows:

- VM_STATE_VALID or VM_STATE_VALID_NOT; translating to whether memory is accessible or not.
- VM_STATE_WRITABLE or VM_STATE_WRITABLE_NOT; translating to whether the program has write permission to the memory segment or not.

The `vmLib` library does not allow the programmer to discriminate between privileged and user modes shown in Table 2.

The IXDP425 platform BSP marks all the memory areas that are illustrated as defined in Figure 2 on page 10 as VM_STATE_VALID and VM_STATE_WRITABLE.

2.1.3 Cacheability and Bufferability

The MMU associates with each section/page of memory a bufferability B-bit and cachability C-bit.

Table 3. Interpretation of Cacheable and Bufferable Bits

C-Bit	B-Bit	Write-Through Cache†	Write-Back Only Cache	Write-Back/Write-Through Cache
0	0	Uncached/Unbuffered	Uncached/Unbuffered	Uncached/Unbuffered
0	1	Uncached/buffered	Uncached/buffered	Uncached/buffered
1	0	Cached/unbuffered	Unpredictable	Write-through cached/ buffered
1	1	Cached/buffered	Cached/buffered	Write-back cached/buffered

† Both write-through and write-back cache policies are explained in section "Write Policy" on page 20.

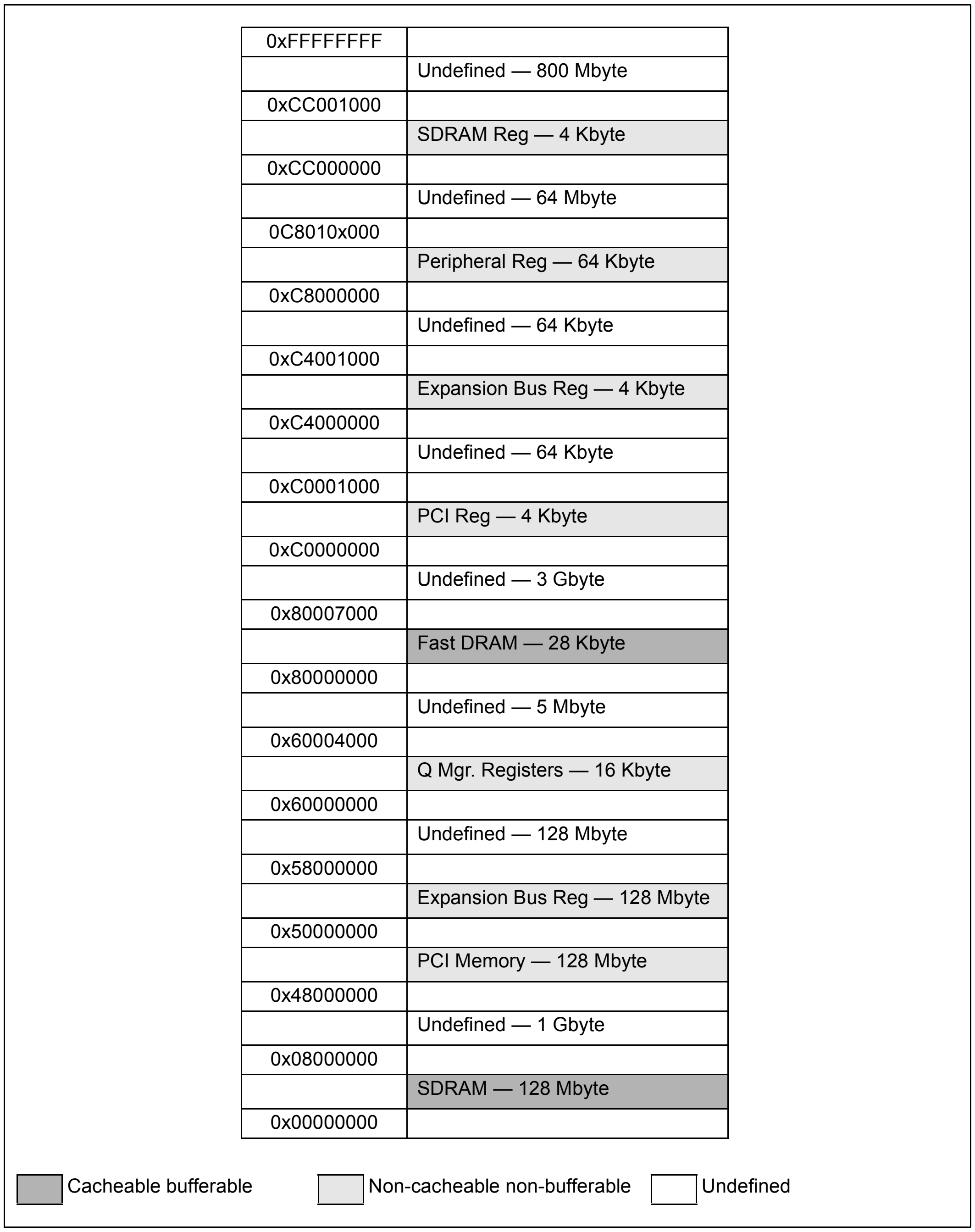
Making a memory area cacheable yields a significant performance gain because the nuMbyteer of external memory accesses, which are long, is reduced.

Writing to a bufferable memory area causes the data to go onto the write buffer, local to Intel XScale core and fast, this allows the Intel XScale core to execute other instructions in its program rather than stall until the write operation is complete.

In order to, safely, designate a memory space as cacheable and bufferable it has to satisfy the following criteria:

- A load from a memory location returns the last value stored into it.
- A store to a memory location has no side effects, e.g., trigger a silicon state machine, except changing its contents.
- Two consecutive reads from a memory location is always guaranteed to return the same value.
- Two consecutive writes to a memory area result in the second value being stored and the first one being discarded.

Figure 3. Intel® IXDP425 Network Processor Development Platform Cachability and Bufferability Status at Startup



0xFFFFFFFF	
	□ Undefined — 800 Mbyte
0xCC001000	
	░ SDRAM Reg — 4 Kbyte
0xCC000000	
	□ Undefined — 64 Mbyte
0C8010x000	
	░ Peripheral Reg — 64 Kbyte
0xC8000000	
	□ Undefined — 64 Kbyte
0xC4001000	
	░ Expansion Bus Reg — 4 Kbyte
0xC4000000	
	□ Undefined — 64 Kbyte
0xC0001000	
	░ PCI Reg — 4 Kbyte
0xC0000000	
	□ Undefined — 3 Gbyte
0x80007000	
	▓ Fast DRAM — 28 Kbyte
0x80000000	
	□ Undefined — 5 Mbyte
0x60004000	
	░ Q Mgr. Registers — 16 Kbyte
0x60000000	
	□ Undefined — 128 Mbyte
0x58000000	
	░ Expansion Bus Reg — 128 Mbyte
0x50000000	
	░ PCI Memory — 128 Mbyte
0x48000000	
	□ Undefined — 1 Gbyte
0x08000000	
	▓ SDRAM — 128 Mbyte
0x00000000	

▓ Cacheable bufferable ░ Non-cacheable non-bufferable □ Undefined

Not all locations in the memory map satisfy the preceding criteria. Therefore, the IXDP425 platform BSP sets up the cachability and bufferability attributes of memory section/page according to the property of each memory space, i.e., the property of the physical hardware residing in memory. Figure 3 shows the initial bufferability and cachability attributes of each memory segment.

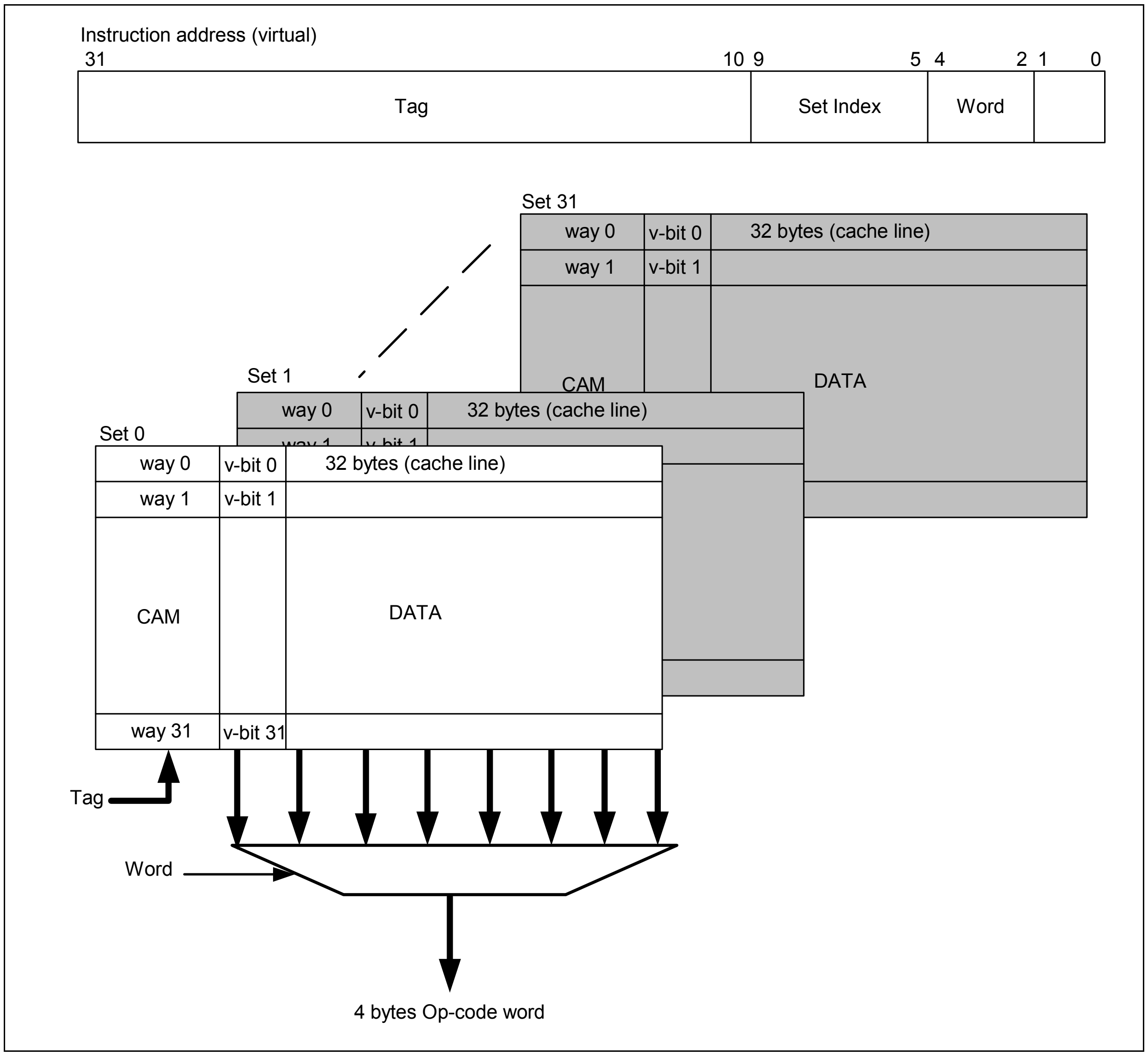
During the life of the program, it may become necessary to allocate SDRAM memory that is neither cacheable nor bufferable. In order to carve out an area of SDRAM and change its attribute to non-cacheable and non-bufferable, the file `xscale_sw/src/include/IxOsCacheMMU.h` provides the macro IX_ACC_DRV_DMA_MALLOC(size).

2.2 Instruction Cache

The Intel XScale core instruction cache minimizes the nuMbyteer of op-code fetches from external memory which in turn delivers better execution performance.

2.2.1 Size and Organization

Figure 4. Instruction Cache Organization



The instruction cache is 32 Kbyte in size and is a 32-way, set associative cache. Each way contains 32 bytes called a cash line and a valid bit to indicate whether the cache line is valid or not. Figure 4 shows the organization of the instruction cache:

- The instruction cache contains 32 sets.
- A set contains 32 ways.
- A way contains one cache line and one valid bit.
- A cache line contains eight words of op code.
- An op-code word contains four bytes.

The instruction cache is virtually addressed, meaning that the virtual address — rather than the physical address — is used in order to determine the set and way to be accessed.

In order to enhance performance and avoid serial searches to find out whether a cache line is resident in cache or not, the instruction cache is structured into 32 sets, bits [9:5] of the virtual address determine the set to be accessed, any given cache line is only ever allowed to reside in one set and one set only. The tag which is bits [31:10] of the virtual address is then submitted to CAM and a cache hit is said to have occurred if and only if the tag submitted matches that of a way in the set. The word field of the virtual address, bits [4:2], is used to select the desired word of the cache line.

The valid bit indicates that the cache line is coherent with main memory or not. By marking a cache line as invalid, the Intel XScale core programmer ensures that the next time op-code is loaded from that same cache line, the cache content will be ignored and the hardware will automatically load the latest version of the cache line from external memory. Once a cache line is fetched from external memory, it is automatically marked as valid.

2.2.2 Fetch Policy

There are two possible outcomes for any op-code fetch. First, the cache line of the op-code word, 4 bytes, is resident in the cache — a “cache hit” being said to have occurred. Second, the op-code’s cache line is not resident in the cache — a “cache miss” being said to have occurred.

Upon a cache miss, the hardware will take the following steps in order to cache in the cache line containing the desired op-code:

1. The cache allocates an internal fetch buffer.
2. The cache sends a 32-byte fetch request to external memory.

   The 32 bytes and the cache line are returned to the fetch buffer at a maximum rate of one clock cycle per 4 bytes. Regardless of the location of the desired op-code within the cache line, the desired word will be returned first. (This feature is called “critical word first.”)

   The latter feature is extremely important because it means that for a cache miss the penalty paid is close to that of a normal external memory read, the fact that the remaining seven words read are done at no cost to the Intel XScale core performance.

Note: The eight words of a cache line are contiguous. Memory is aligned on a 8-word basis, i.e., equal to cache line’s size.

3. As soon as the first word (4 bytes) are received, they are forwarded to the instruction decoder for execution.

4. When all 32 bytes are received, the newly fetched cache line is updated into the cache. The cache line updated is chosen for replacement is a result of the round-robin algorithm used by the instruction cache. (See section "Cache-Line Replacement Policy" on page 17.)

2.2.3 Cache-Line Replacement Policy

The round-robin cache-line replacement algorithm operates on a per-set basis. As explained in "Size and Organization" on page 15, the cache is organized into 32 sets — each set contains a pointer to a way, i.e., a cache line.

At startup, the replacement pointer of each set is pointing at the last way, Way 31. Once a cache line is written into Set 31, the pointer is moved to Way 0 and so on. The movement of a set pointer has no consequence on the pointers of the other 31 sets.

Note: The pointer is moved to Way 0, providing that Way 0 is not locked into cache, if it is the pointer skips and points to Way 1.

2.2.4 Locking

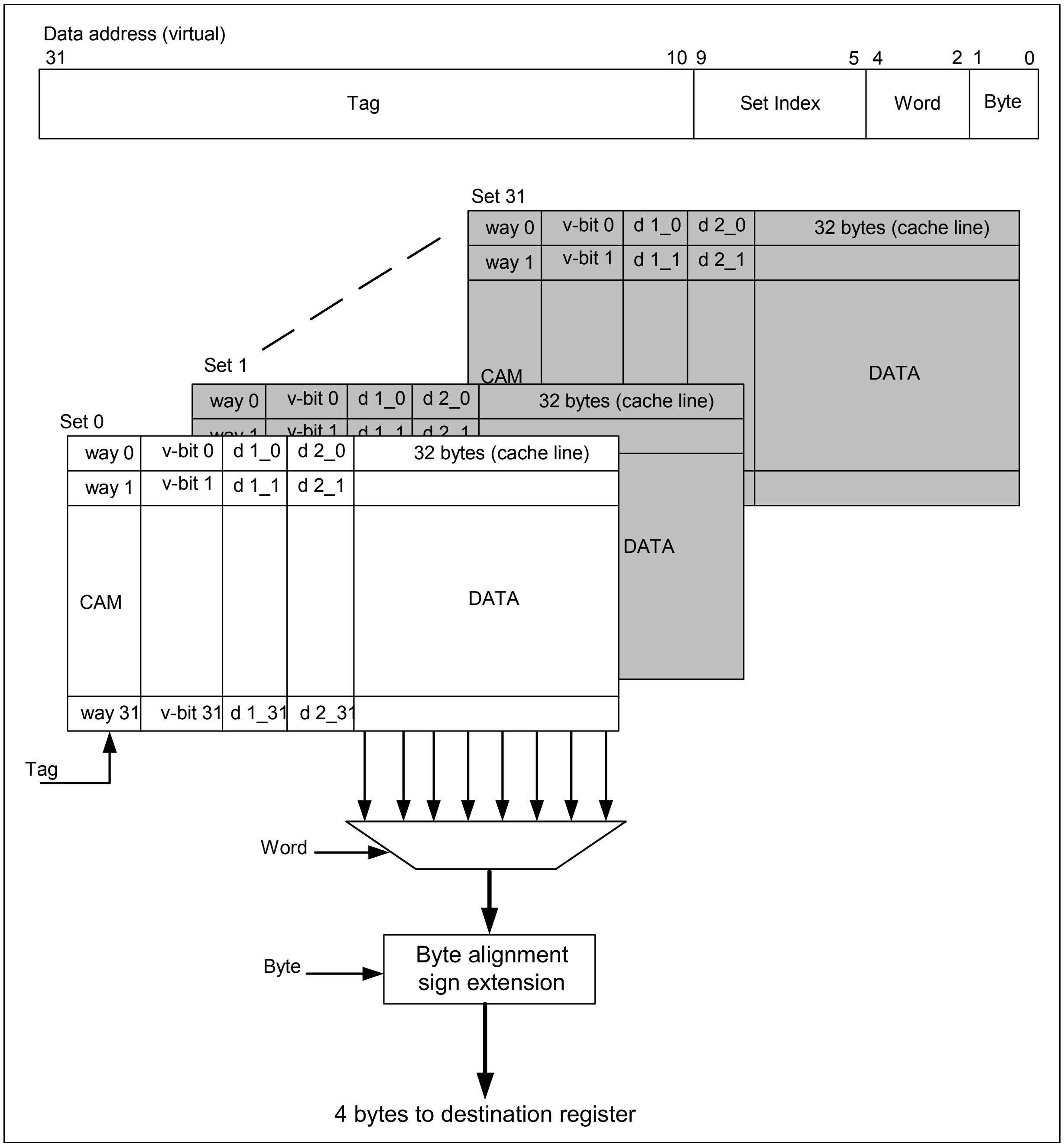
The instruction cache allows the programmer to lock up to 28 ways per set, effectively given the developer the ability to turn the instruction cache into a 28-Kbyte, fast, read-only memory. Ways 0 to 27 are lockable while 28 to 31 are not. This feature can be useful in the event where performance becomes an issue, and certain code routines must be optimized and thus be permanently resident in cache.

2.3 Data/Mini-Data Cache

The Intel XScale core contains two caches for storing data: a data cache and a mini-data cache. These two caches contribute to performance enhancement by reducing the time the Intel XScale core has to access external memory in order to load or store data.

2.3.1 Size and Organization

Figure 5. Data Cache Organization



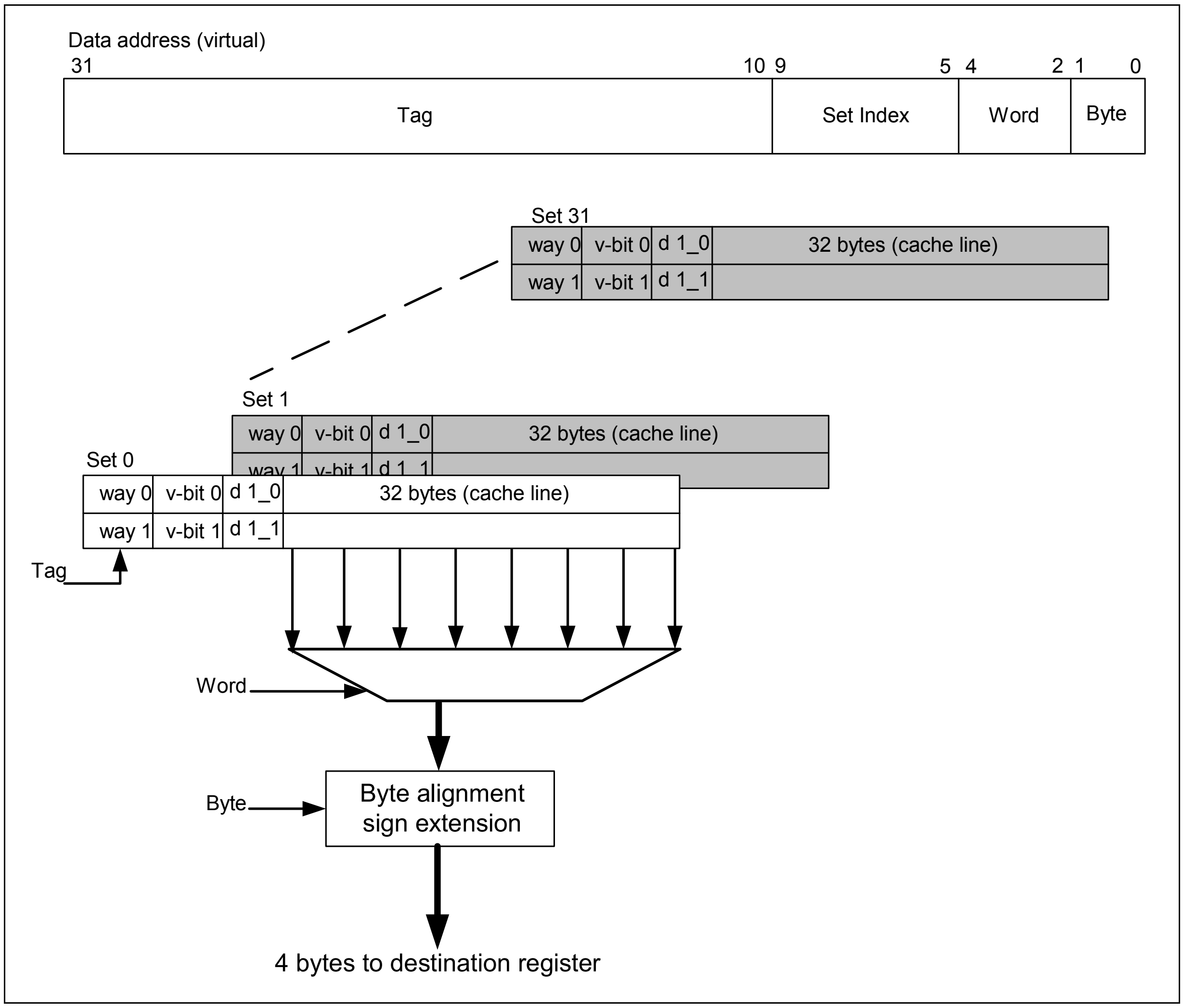
The data-cache is 32 Kbyte in size and is a 32-way, set associative cache, each set contains 32 ways. Each way contains a 32-byte cache line, a valid bit, and two dirty bits. One of the dirty bits is for the upper 16 bytes and the second dirty bit is for the lower 16 bytes of the cache line.

Figure 5 illustrates the organization of the data cache:

- The data cache contains 32 sets.
- A set contains 32 ways.
- A way contains a cache line, a valid bit, and two dirty bits.
- A cache line contains 32 bytes of data.

In order to enhance performance and avoid searches, the cache submits both the set index and tag fields of the virtual address to CAM in order to determine whether or not the cache line is currently present in cache.

Figure 6. Mini-Data Cache Organization



The mini-data cache is 1/16 the size of the data cache, i.e., 2 Kbyte, and is a 2-way, set associative cache, each set contains 2 ways and each way contains a cache line of 32 bytes, and a valid bit. Similar to the data cache, the mini-data cache line contains two dirty bits one for the upper 16 bytes of the cache line and the other for the lower 16 bytes of the cache line.

In summary:

- The mini-data cache contains 32 sets.
- A set contains 2 ways.
- A way contains one cache line, one valid bit, and 2 dirty bits.
- A cache line contains 32 bytes of data.

The arrangement of mini-data cache is shown in Figure 6.

The mini-data cache is added to the Intel XScale core to hold data that is contiguous and likely to be modified once and then written back to main memory. Using mini-data cache for this type of data enhances performance and lessens the load on the data-cache.

The IXDP425 platform's BSP does not associate any virtual address of the Intel XScale core memory map with the mini-data cache.

2.3.2 Read Policy

"Data Cache Flow Diagrams" on page 28 illustrates the behavior of the data cache and mini-data cache's, on a data load. The operations shown in that section are performed by the caches' hardware and require no software intervention.

Upon a cache miss, the "critical word first" feature ensures that the Intel XScale core does not stall while the whole cache line is being clocked out of external memory into the caches' fill buffer.

2.3.3 Write Policy

A cache write cycle takes different paths depending on the write policy chosen for the memory section/page set in the MMU. (See "Cacheability and Bufferability" on page 12.) If a memory area is marked as write-through in the MMU, every write cycle will cause both the cache and the external memory to be updated.

However, if the memory segment is marked as write-back in the MMU, a write to that memory location will cause:

- The new data to be stored in the appropriate location in cache, without accessing external memory.
- The dirty bit associated with the cache line half that has been modified to be set.

When the round-robin purging algorithm for the set chooses the line marked dirty to be purged back to external memory to make room for a new cache line, the dirty 16 bytes — previously marked dirty — are written back to external memory. This is done to keep cache and external memory coherent.

2.3.4 Cache Line Replacement Policy

Both data and mini-data caches' line replacement policies are similar to that of the instruction cache. (See section "Cache-Line Replacement Policy" on page 17.)

2.3.5 Cache Invalidate and Cleanup

The programmer can invalidate a data cache line by invoking the macro IX_ACC_DATA_CACHE_INVALIDATE(addr, size) in the header file `xscale_sw/src/include/IxOsCacheMMU.h`. Invalidating a cache line will result in the cache line to be loaded from external memory the next time the Intel XScale core accesses it.

The programmer can also order a cleanup of a data cache line by invoking the macro IX_ACC_DATA_CACHE_FLUSH(addr, size) in the file `xscale_sw/src/include/IxOsCacheMMU.h`.

Cleaning “flushing” a cache line causes it to be written back to external memory.

Note: Flushing data to external memory is not necessary when a write-through cache policy is in use.

2.3.6 Locking

The data cache provides a locking mechanism that is similar to that of the instruction cache, discussed in “Locking” on page 17. Data cannot be locked into mini-data cache.

3.0 Handling Data Traffic

3.1 Facts

The programmer must be aware of some facts when porting software from the Intel® IXP22x processor — a device that does not have an MMU — to a chip that has a MMU, such as the Intel® IXP42X product line and IXC1100 control plane processors.

The facts can be summarized as follows:

- All Intel XScale core memory accesses are subject to Intel XScale core MMU operation.
- All memory accesses from Intel XScale core will be translated to a physical address by the MMU.
- An NPE only addresses physical memory, and has *no* notion of Intel XScale core virtual memory.
- NPE will only be able to read/write data in physical SDRAM memory.

  If, at time t, memory location M has the value D1 in physical SDRAM memory and D2 in cache, the NPE will read the value D1.
- If, at time t, memory location M is present in cache and has the value D1, at time t+1 the NPE stores the value D2 into M. At time t+2 — when the Intel XScale core reads the value in M — it will read D1.

3.2 Sharing Data Between Intel XScale® Core and NPE on the Intel® IXDP425 Network Processor Development Platform

In light of the facts listed in "Facts" on page 21, this subsection explains how data is shared between the Intel XScale core and the NPEs in a safe manner on the IXDP425 platform.

Intel® IXP42X product line and IXC1100 control plane processors contain one Intel XScale core and three NPEs — each one being a processor in its own right. Among other features, the Intel XScale core has MMUs and caches to speed up access to both op-code and data. The operation of the MMUs and the cache are transparent to the programmer, as long as the data manipulated is local to the Intel XScale core and not being shared with the other processors on the Intel® IXP42X product line and IXC1100 control plane processors.

Sending and receiving traffic involves the sharing of data by the four processors on the Intel® IXP42X product line and IXC1100 control plane processors. It is during this sharing of data that the Intel XScale core programmer needs to be aware of the MMUs and caches.

The following discussion will highlight the steps that the programmer of the Intel XScale core core needs to take in order to guarantee correct transmission and reception of traffic.

In the sub-sections that follow, the mbuf buffer format receives special attention because it is the structure adopted to carry traffic on the Intel® IXP42X product line and IXC1100 control plane processors.

3.2.1 Allocating Memory for Send and Receive Operations

The 128 Mbyte of SDRAM on the IXDP425 platform is used to store decompressed code from flash memory, to store stacks for different threads running on Intel XScale core, program data, as well as to store buffers that will be shared with the NPEs.

At boot time, the BSP sets the attribute of the whole 128 Mbyte space as valid, cacheable, and read/writable. The idea is to improve the Intel XScale core performance by reducing the number of times it has to go to external memory in order to load or store data.

The policy adopted for Intel® IXP4XX Product Line Software Release 1.0 codelets was to allocate memory that stores the mbuf-related structure using the IX_ACC_DRV_DMA_MALLOC(size) provided in the file `xscale_sw/src/include/IxOsCacheMMU.h`. The macro allocates the necessary memory space from SDRAM and sets the cacheable attribute of the memory segment/page in the MMU to uncacheable.

Marking a memory segment/page as uncacheable means that stores to it always update the physical memory, and that loads from it cause the Intel XScale core to read from external memory.

Among the first steps of creating an mbuf pool that is non-cacheable, is the creating of memory space to hold both the Mbytelk/clbk structures and the clusters. The developer must ensure that memory from which the Mbytelk, clblk, and cluster structures are carved — using netBufLib's netPoolInit function — is allocated using the macro IX_ACC_DRV_DMA_MALLOC(size).

Figure 7. Example Code for Creating a Cache-Safe Mbuf Pool

```
/* Pointer to the pool */
NET_POOL_ID poolId;

/* Declare Mbytelk/clBlk config structure */
M_CL_CONFIG mClConfig =
{
/*      MbytelkNum     clBlkNum   memArea    memSize */
          64,            64,        0,         0
};
/* Declare cluster description table */
CL_DESC clDescTbl[] =
{
/*      cluster size      clNum      memArea    memSize */
          {1024,          64,         0,         0}
};
int createMbufPool(void)
{
  /* Get the nuMbyteer of elements in the cluster description table */
  int descTblNumEnt = NELEMENTS(clDescTbl);
  /* Set the size of the Mbytelk/clBlk storage space */
  mClConfig.memSize = (mClConfig. ClBlkNum * (M_BLK_SZ + sizeof(long))) +
                                            (mClConfig. ClBlkNum * CL_BLK_SZ);
  /* Allocate memory for Mbytelk/clBlk using macro to make sure it is not cacheable */
  mClConfig.memArea = (char*) IX_ACC_DRV_DMA_MALLOC(mClConfig.memSize);

  /* Set the size of the clusters storage space */
  clDescTbl[0].memSize = (clDescTbl[0].clNum * (clDescTbl[0].clSize + sizeof(long)));
  /* Allocate memory for clusters using the macro to make sure the memory space is not
cacheable */
  clDescTbl[0].memArea = (char*) IX_ACC_DRV_DMA_MALLOC(clDescTbl[0].memSize);

  /* Carve the Mbuf pool */
  return (netPoolInit(&poolId, & mClConfig, clDescTbl, descTblNumEnt, NULL));
}
```

The example shown in Figure 7 illustrates how to create a cache-safe mbuf pool. Creating cache-safe mbuf pools ensures coherence between memory accessed by the Intel XScale core and NPEs.

3.2.2 Intel XScale® Core Submission of Data to the NPE for Transmission

The combination of the Intel XScale core, AHB queue manager, and the NPEs gives the Intel® IXP42X product line and IXC1100 control plane processors the ability to rapidly process packets. The gain in performance arises from the fact that the Intel XScale core only has to post the *address* of the buffer to be transmitted to the AHB queue manager, which is effectively a set of hardware queues. The NPE uses the address posted to locate, in memory, the frame to be transmitted.

The programmer is faced with two problems:

- Intel XScale core MMUs' virtual-to-physical address translation
- SDRAM and cache coherence

Before submitting a mbuf to the queue manager, the Intel XScale core developer must ensure that the *physical address* is posted and not the virtual address. In the context of the NPEs memory space the Intel XScale core's virtual addresses could be garbage. The macro IX_MMU_VIRTUAL_TO_PHYSICAL_TRANSLATION(addr) should be used to convert the address of the mbuf, prior to its submission to the AHB queue manager.

Figure 8. Example Code: Preparing Mbuf to be Submitted to Queue Manager

```
struct Mbuf* prepareToSendChainedMbufToNpe(struct Mbuf *pBuf)
{
   char *pData;
   struct Mbuf  *pChain = pBuf;
   struct Mbuf *tempBuf;
   if (pBuf == NULL)
      return pBuf;
   /* Convert the address of the Mbuf passed in */
   IX_MMU_VIRTUAL_TO_PHYSICAL_TRANSLATION(pBuf);
   while (pChain)
   {
      /* Save the address of the next buffer in the chain and the pointer to the data */
      tempBuf = pChain->m_next;
      pData = pChain->m_data;
      /* Convert the relevant addresses */
      IX_MMU_VIRTUAL_TO_PHYSICAL_TRANSLATION(pChain->m_next);
      IX_MMU_VIRTUAL_TO_PHYSICAL_TRANSLATION(pChain->m_data);

     /* Flush the cache lines associated with the Mbuf header into external memory */
     IX_ACC_DATA_CACHE_FLUSH(pChain->MbytelkHdr, sizeof(pChain->MbytelkHdr));
      /* Flush the cache lines associated with cluster into external memory */
      IX_ACC_DATA_CACHE_FLUSH(pData, pChain->m_len);

      /* next */
      pChain  = tempBuf;
   }
   return pBuf;
}
```

"Allocating Memory for Send and Receive Operations" on page 22 emphasized the fact that Intel® IXP4XX Product Line Software Release 1.1 codelets adopted the use of IX_ACC_DRV_DMA_MALLOC(size) to allocate non cacheable memory in order to make sure that the cache and the SDRAM are always coherent. Based on this, the memory coherence problem listed in point number two, above, should not pose itself. However, a customer might decide not to have non-cacheable memory and, therefore, set the IX_ACC_DRV_DMA_MALLOC(size) macro to an empty string. In this last case, the coherence-related issues arise.

The way around the memory coherence problem is to make sure that the macro IX_ACC_DATA_CACHE_FLUSH is invoked prior to submitting the buffers to the NPE. Currently, in Intel® IXP4XX Product Line Software Release 1.1, the IX_ACC_DATA_CACHE_FLUSH is defined as an empty string, so it does nothing. The rationale behind invoking this "empty" macro is to make sure that code can be ported easily, in case future projects decide not to implement the IX_ACC_DRV_DMA_MALLOC macro and implement IX_ACC_DATA_CACHE_FLUSH macro to flush data from cache into external memory.

Figure 8 gives an example of a function used to prepare a mbuf to be submitted to the NPEs. The following are four important points, related to Figure 8, must be considered:

- As far as virtual to physical address translation is concerned, there are more pointers inside a mbuf structure than is shown in Figure 8.

  Figure 8 assumes that the NPE will only use the m_next and m_data element pointers of the mbuf, this has to be verified at design time.
- Within the Intel XScale core, a policy has to be adopted — between access layer software and application software — as to which component converts and flushes what fields of the mbuf

chain. Failure to do so could cause a virtual address to be converted to a physical address twice which will result in erroneous operation.

- Similar to point number one, Figure 8 assumes that the NPE only operates on the MbytelkHdr and the cluster field. This assumption must be verified at the design phase of the project.
- The example listed in Figure 8 does not cater for the issue of endianess that occurs between the Intel XScale core and the NPE. This issue is *very important* but is considered beyond the scope of this technical note.

3.2.3 Intel XScale® Core Retrieving Data from the NPE for Reception

On the receive path — for example, from the NPE to the Intel XScale core — the developer is exposed to the mirror issues of those mentioned in "Intel XScale® Core Submission of Data to the NPE for Transmission" on page 23, namely:

- The mbuf addresses in the reception path queues are physical addresses.
- The Intel XScale core might have an out of date copy of the data being received in its cache.

Prior to acting on a received mbuf, the Intel XScale core software must convert the physical address to its equivalent physical address. The physical to virtual address translation can be achieved through the IX_MMU_PHYSICAL_TO_VIRTUAL_TRANSLATION(addr).

Figure 9. Example Code: Processing an Mbuf Received from NPE

```
struct Mbuf* prepareToRcvChainedMbufFromNpe(struct Mbuf *pBuf)
{
   struct Mbuf  *pChain;
   struct Mbuf *tempBuf;
   if (pBuf == NULL)
      return pBuf;
   /* Convert the address of the Mbuf passed in */
   IX_MMU_PHYSICAL_TO_VIRTUAL_TRANSLATION(pBuf);

   pChain = pBuf;
   while (pChain)
   {
     /* Invalidate the Mbuf header prior to operating on it */
     IX_ACC_DATA_CACHE_INVALIDATE(pChain, sizeof(pChain->MbytelkHdr));
      /* Invalidate the cluster if present in the cache */
      IX_ACC_DATA_CACHE_ INVALIDATE (pChain->m_data, pChain->m_len);

      /* Save the address of the next buffer in the chain */
      tempBuf = pChain->m_next;

      /* Convert the relevant addresses */
      IX_MMU_PHYSICAL_TO_VIRTUAL_TRANSLATION (pChain->m_next);
      IX_MMU_PHYSICAL_TO_VIRTUAL_TRANSLATION (pChain->m_data);

      /* next */
      pChain  = tempBuf;
   }
   return pBuf;
}
```

Software must also invalidate the version held in cache of the data received from the NPEs, this is achieved by invoking the IX_ACC_DATA_CACHE_ INVALIDATE macro.

Strictly speaking, if the memory used for sharing data with the NPE has been allocated — using the macro IX_ACC_DRV_DMA_MALLOC — invalidating the cache line for this data is not necessary because the memory is marked as non-cacheable. The implementation of the IX_ACC_DATA_CACHE_ INVALIDATE macro on the IXDP425 platform is an empty string.

The idea behind invoking the IX_ACC_DATA_CACHE_ INVALIDATE macro — prior to acting on data from the NPE — is to ensure that, in the event that future projects decide not to implement the IX_ACC_DRV_DMA_MALLOC macro and thus not to have non-cacheable memory is minimal. Indeed in such a case, the impact will be contained in the file `xscale_sw/src/include/IxOsCacheMMU.h` where the macro IX_ACC_DATA_CACHE_ INVALIDATE will have to be implemented.

Figure 9 provides example code of how to handle data received from the NPE. Similar to the warning in "Intel XScale® Core Submission of Data to the NPE for Transmission" on page 23, it is crucial that the NPE code and the Intel XScale core are in tune as to what mbuf fields shall be used by each party.

3.2.4 Options to Ensure Memory Coherence

The macros IX_ACC_DRV_DMA_MALLOC, IX_ACC_DATA_CACHE_FLUSH, and IX_ACC_DATA_CACHE_ INVALIDATE allow the programmer to mark memory segments as noncacheable, force the content of cache to be written to external memory, and invalidate cache entries, respectively.

This paper *stresses* the use of those three macros in order to achieve memory coherence, and minimize the impact of changing the memory coherence strategy. Examples are listed in Figure 8 on page 24 and Figure 9 on page 25.

If the strategy adopted is to enable the IX_ACC_DRV_DMA_MALLOC macro and disable the IX_ACC_DATA_CACHE_FLUSH and IX_ACC_DATA_CACHE_ INVALIDATE macros — as is the case on the IXDP425 platform — the penalty associated with external memory access will be incurred by the Intel XScale core every time it reads or writes to this memory. In order to increase performance, the software designer has to estimate the frequency at which the Intel XScale core will need to read/write to the shared memory area before submitting it to the NPE. The higher the access frequency, the less viable this strategy becomes.

However, if the strategy is to disable the IX_ACC_DRV_DMA_MALLOC macro and enable the IX_ACC_DATA_CACHE_FLUSH and IX_ACC_DATA_CACHE_ INVALIDATE macros, the penalty of accessing external memory is only incurred upon cache invalidation or flushing.

This strategy holds performance benefits if the shared area is likely to be accessed a considerable number of times by the Intel XScale core prior to submitting it to the NPE. However, if the Intel XScale core accesses this memory a minimal number of times, there is not much difference between the two strategies. In this last case, it might be simpler to mark the shared memory segment as noncacheable.

3.3 Best Practices for Intel XScale® Core-NPE Data Sharing

- In order to avoid memory contention and erroneous operation, data must not be accessed, once it has been submitted to the NPEs.
- Intel XScale core software must only use pointers passed in by the NPEs after they have been translated from physical to virtual addresses. This includes pointer-de-referencing operations.

- Non-cacheable memory must only be accessed when absolutely necessary. Excessive access to non-cacheable memory will negatively impact Intel XScale core performance.
- It is recommended that functional specifications clearly state when and why the Intel XScale core accesses non-cacheable memory.

4.0 Data Cache Flow Diagrams

Figure 10. Data Cache Read Cycle Flow Diagram

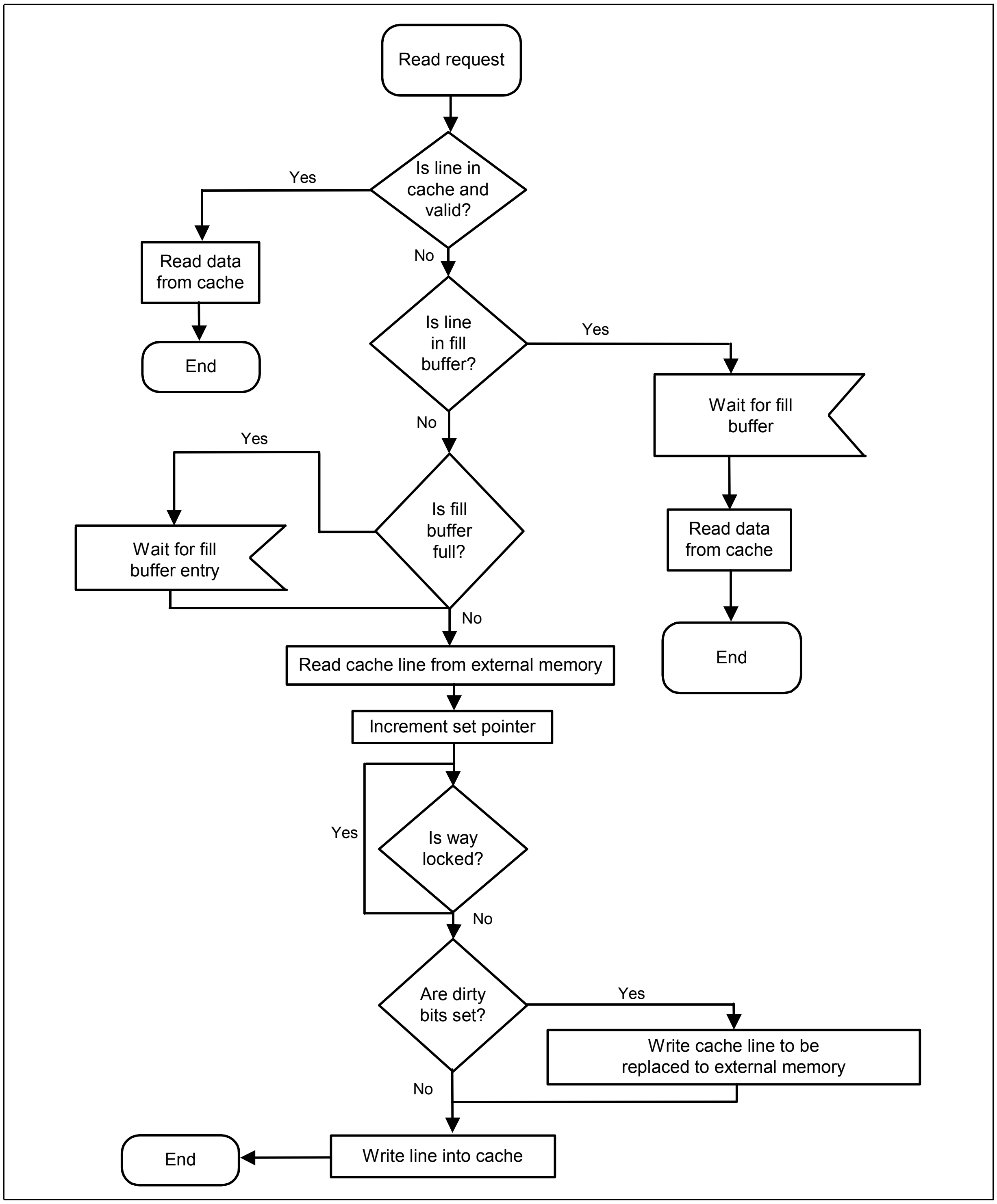
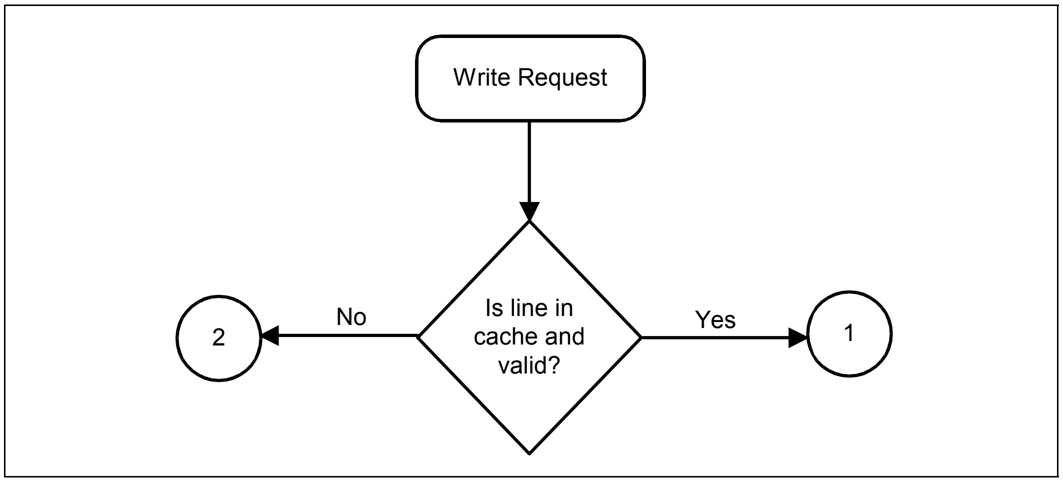


Figure 11. Data Cache Write Cycle Flow Diagram: One of Three



1. The "1" event is continued in Figure 12.
2. The "2" event is continued in Figure 13 on page 30.

Figure 12. Data Cache Write Cycle Flow Diagram: Two of Three

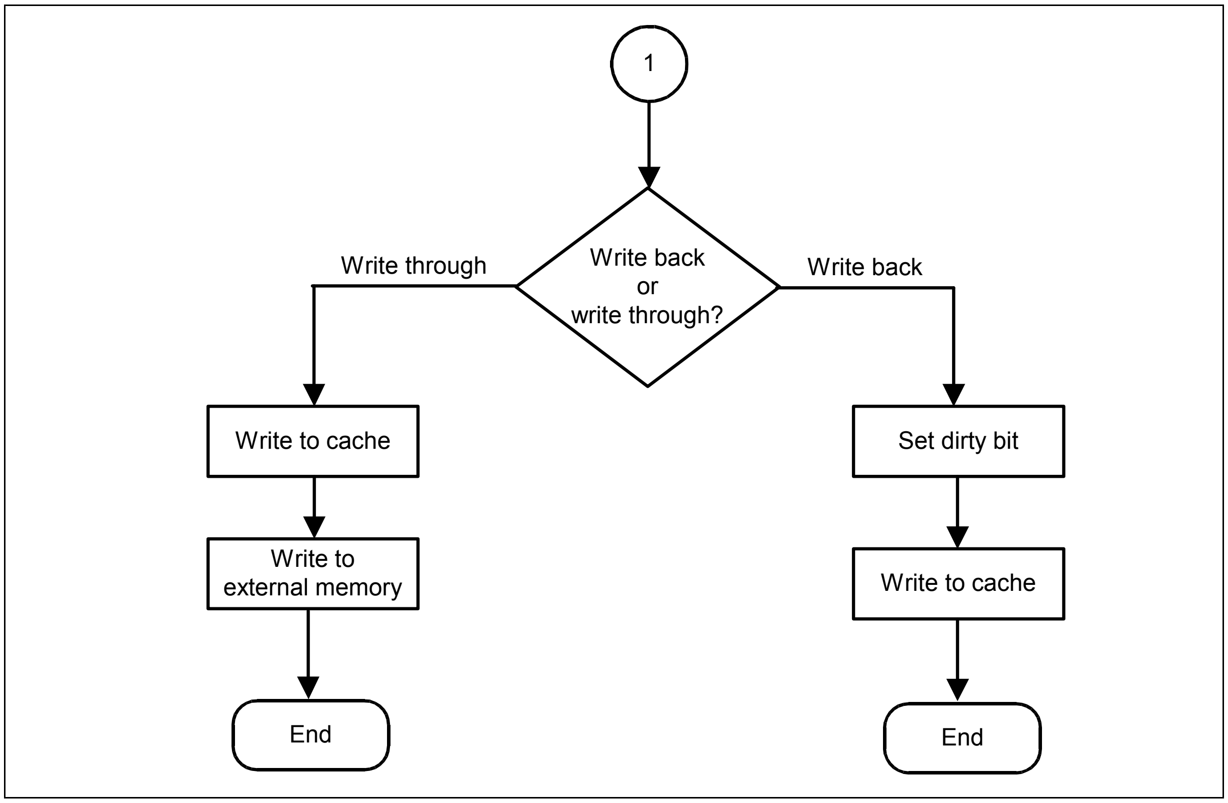
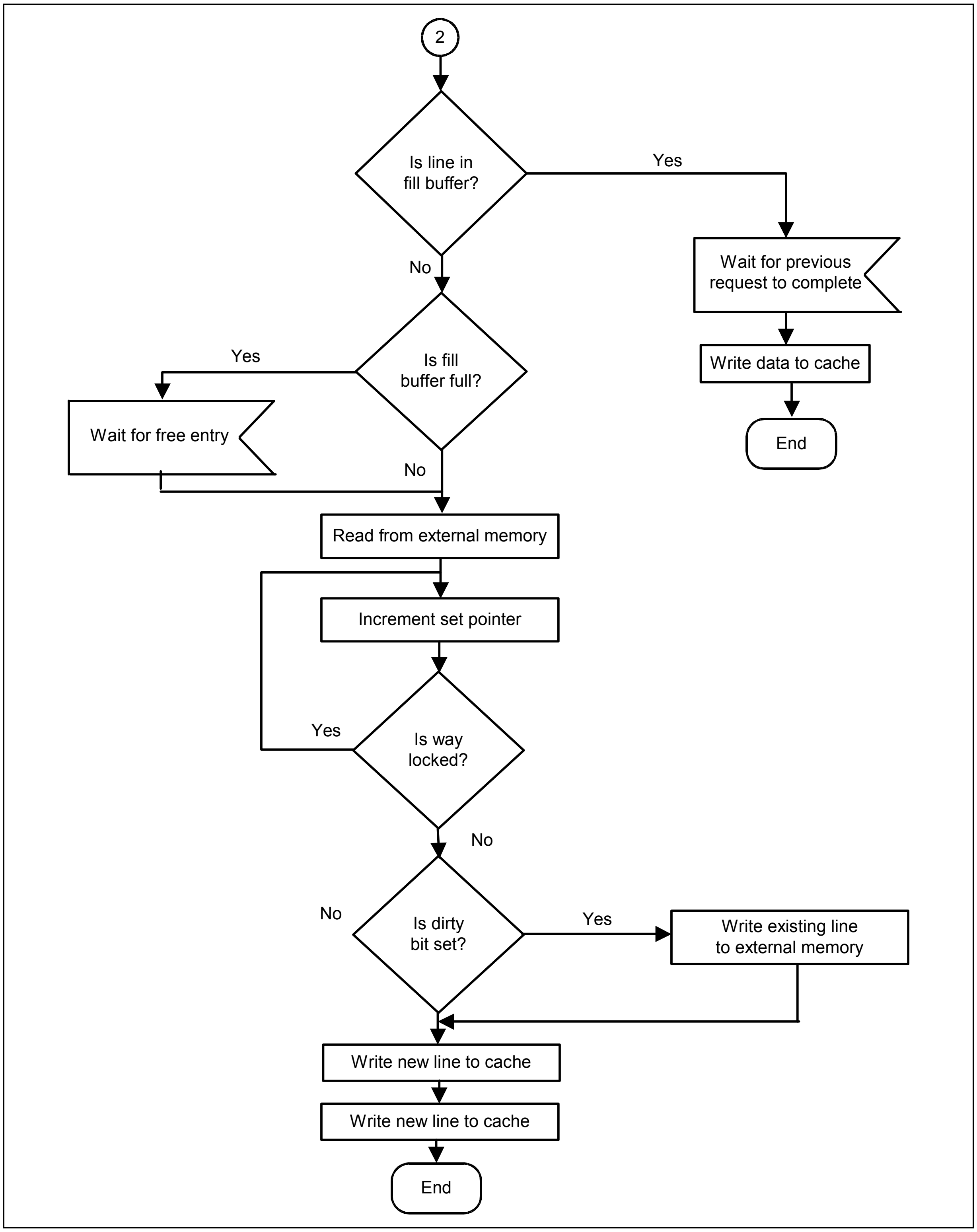


Figure 13. Data Cache Write Cycle Flow Diagram: Three of Three


2
Is line in fill buffer?
Yes
No
Wait for previous request to complete
Write data to cache
End
Yes
Is fill buffer full?
Wait for free entry
No
Read from external memory
Increment set pointer
Yes
Is way locked?
No
No
Is dirty bit set?
Yes
Write existing line to external memory
Write new line to cache
Write new line to cache
End