SiCortex

To: Distribution

From: Matt Reilly

Date: 14-Feb-2008

Subject: Overview of SiCortex Cache Coherence Architecture

As we start on the IceT development and verification it has become clear that we need a brief primer on the general workings of the SiCortex cache coherence protocol. This memo will describe a few of the more common operations supported by the protocol, with a specific focus on the proposed implementation in the IceT processor core.

This memo is not intended as a verification reference: it is a primer. The protocol is described in excruciating detail in the SiCortex Ice9 Specification. The IceT core makes use of the same protocol as implemented in Ice9, with a few performance improvements. IceT protocol flows are described in detail in the Mbox and Cbox chapters of the IceT spec. The IceT specification should be viewed as the ultimate authority on actual protocol flows.

Cache Coherence: General

The SiCortex cache coherence protocol is based on the MESI (MODIFIED, EXCLUSIVE, SHARED, and INVALID) four state scheme. There are dozens of rules and procedures that describe the behavior and legal transaction paths for the protocol, but there are a few fundamental aspects:

- The protocol allows read-only copies of a block to exist in multiple caches at the same time. They will all be held in the SHARED state.
- If a block is to be written, exclusive ownership of the block must be

acquired by the processor that updates the block.

- Writeable blocks are held in the EXCLUSIVE state (bits are the same as in main memory) or the MODIFIED state (bits have been changed since last read from main memory).
- Blocks in the EXCLUSIVE or MODIFIED state exist in only one processor's cache at any time.
- Memory transfers occur one cache-block at a time.
- A cache block contains 64 bytes.

The protocol also supports block transfers to and from blocks that are currently resident in the EXCLUSIVE or MODIFIED state in a processor's cache.

Coherence is maintained by sequencing all cache-miss operations through one of the two coherence engines (COH) on an Ice9 or Twice9 chip. Odd block addresses (bit 6 of the physical address is set) are handled by the odd coherence widget (COHO), while even addresses are routed to the even widget (COHE). A COH examines each memory fetch request and performs one of three operations:

1. It forwards the physical address to the attached DRAM controller.
2. It forwards the physical address to a processor that is believed to have a cached copy of the block.
3. It places the request in a queue for later processing when a currently active request against the same address has been completed.

Additionally, the COH also forwards DRAM write back operations directly to the DRAM controller.

All operations initiated by a processor are tagged with a unique identifier called a transaction identifier or TID. Each processor in the Twice9 chip is assigned four TIDs for its sole use.[2]

Processors are connected to the COH units via a time multiplexed, daisy-chained, multidrop, multistage synchronous switch called the CSW. Access to the CSW is moderately fair and starvation proof.

[2]Each processor also owns a fifth TID to identify a special ownership transfer operation called a WRSTRANS. If this is important to you, you should be reading the IceT specification.

All IceT processors contain three cache units:

- A first level 32KB four-way set associative D-cache.
- A first level 32KB four-way set associative I-cache.
- A second level 256KB four-way combined (I and D) cache.

The L1 D-cache is a proper subset of the L2 cache. All cache references and fills are processed by the appropriate first level cache, then passed through the second level cache.

A Simple Load Miss

Let's follow the execution path of a simple load instruction `LD R5, 0(R6)`.[3]

Step 1

The IceT core will calculate an effective address by adding 0 to the contents of integer register R6. This virtual address will be split into a page-offset field (the low 16 bits: for now, assume that the smallest supported page size is 64KB) and the virtual page number. Bits $[12:6]$ of the page offset are used to index the cache tag RAM, and bits $[12:3]$ are used to index each of the four cache way arrays. At the same time, the upper bits of the address pass through the translation look-aside buffers to be translated to a physical page number.

Step 2

The processor compares the resulting physical address to the tag fields corresponding to each of the four ways at the selected index. If none of the tag entries match the translated address, the access will miss in the L1 cache. The processor will enter the address in the Miss Address File (MAF) and send the physical address on to the L2 cache.

Step 3

The L2 cache unit will lookup the physical address in the L2 tag arrays and fetch data from each of the four ways at the appropriate L2 index. If the access hits in the L2, the block of data is returned to the L1 cache, the appropriate bits are written to R5, and the MAF entry is freed up. However, if the access misses in the L2 cache, it must be sent to the

[3]This example assumes that the page containing the address in question is marked writable in its page table entry.

appropriate coherence unit for fulfillment.

Step 4

The L2 to CSW interface checks bit 6 of the physical address, and if set, sends a RDEX (for ReaD EXclusive) request to COHO (the odd COH). Otherwise the processor sends a RDEX request to COHE. Assume that bit 6 is zero.

Step 5

COHE will receive the RDEX request for address X, forward X to the DRAM controller, and lookup the address in its shadow copy of each processor's L2 tags. If the address is found in one of the shadow tag arrays, the COH must forward a probe request on **A**'s behalf to the current owner of the block. I'll describe this case later.

Step 6

Assume for the moment that X is not in any processor's cache. The COH has already sent the address to the DRAM controller. At some later time, the data will return from the DRAM and the COH will send the data along to processor **A**and clean up any residual state that it has maintained for this transaction.[4]

Step 7

Processor **A**will write the data into its L1 cache, write the appropriate bits to register R5, remove the corresponding entry from the MAF, and the transaction will complete.

Moving Writable Data Between Caches

But now imagine that in **Step 5** the COH had found address X in the EXCLUSIVE state in its copy of processor **B**'s L2 tags.

Step 6F

In this case, the COH will forward a PRBWIN (PRoBe Writeback-INvalidate) request on **A**'s behalf to processor B.

[4]The "residual state" is maintained to allow chaining of transactions that are pending to the same address. See below, under "Sharing Data Between Caches."

Step 7F

Processor **B** will enqueue the PRBWIN request in its local probe queue where it will initiate an L1 cache lookup. If the L1 lookup results in a hit against the L1, the L1 copy of the block will be invalidated, and the data will be sent to the CSW interface via the L2 pipeline.

Step 8F

If the probe access missed in the L1, it will be looked up in the L2. If the probe hits in the L2, the block will be read out of the L2, sent to the CSW interface, and invalidated. If the probe hit in the L1, the data forwarded from the L1 will be sent along to the CSW interface. If the probe misses in both the L1 and L2, the forward probe has become involved in a "ships-passing-in-the-night" event and processing will continue with **Step 9N** below.

Step 9F

If the probe access hit in either of processor **B**'s caches, **B** will send the block of data to processor **A**through the CSW.

Step 10F

At some later time, **A**will grab the data from the CSW, write it to its L1 cache, write the appropriate bits to R5, remove the corresponding entry from the MAF, and send a PRBDONE command to COHE to complete the transaction.

Step 11F

COHE will eventually receive the PRBDONE command from processor **A** and clean up any residual state that it has maintained for this transaction.

Step 9N

If the chain of events leads to this step, a forwarded probe request found that the data that was supposed to be in this cache has been evicted. Because of a whole host of rules around sequencing of operations in the COH units, we know now that the data must reside in or be on its way to main memory. Processor B, having found no match for the probe, sends a PRBNOHIT (PRoBe-NOHIT, oh, you already knew *that*) to processor **A**.

Step 10N

Processor **A**, on receiving the NOHIT, now knows that the block containing X must be in the DRAM or on the way to DRAM. **A** then sends a RDRTRY (ReaD ReTRY) for address X to COHE where the address X is sent directly to the DRAM and further processing continues with **Step 6** above.

Sharing Data Between Caches

So far, we've assumed that a block of data exists in only one cache at a time. We need some mechanism for sharing read-only copies of data between processors.[5] The most obvious kind of read-only data are the bits that make up an instruction stream. In fact, this was the only data that was shared in the Ice9 implementation. IceT supports shared data for data-stream accesses as well. (The decision as to whether the processor should cache a block SHARED or EXCLUSIVE is governed by bits in the page table entry, the type of instruction that caused the initial cache miss, and other factors documented in the Mbox chapter of the IceT spec.)

Consider now the load instruction `LD R5, 0(R6)` executed in processor **A** where the address X in R6 is marked as non-writable. In this sequence X is currently residing in the SHARED state in processor **B**'s cache.

The sequence of actions begins in the same manner as in **Steps 1** through **5**, except that processor **A** emits a RDS (ReaD Shared) request in **Step 4**.

Step 6S

The COH will find that X is cached in **B**'s L2 cache in the SHARED state. As a result, the COH will forward a PRBSHR request to **B** on behalf of processor **A**.

Step 7S

Processor **B** will process the incoming PRBSHR request by looking X up in its L1 tag array. If the address hits in the L1 cache, the data is forwarded to the CSW but it is left in the SHARED state in the L1 cache.

From here, the processing of the transaction proceeds as in **Steps 8F** and

[5] We don't support sharing read-write data between processors. Some cache protocols do, but they tend to be very hard to implement, and even harder to implement correctly. Given our expected usage model, it is hard to see a significant performance advantage to such protocols.

on with the exception that the data found in either processor **B**'s L1 or L2 is left in the SHARED state.

SHARED accesses against blocks that are cached in EXCLUSIVE state in another processor, or EXCLUSIVE accesses against blocks that are cached SHARED elsewhere are handled by a more complex sequence of events that are described in the IceT Mbox and Cbox chapters.

Victimization

So far I've described reads to read-only data (reads to SHARED state) and reads to read-write data (reads to EXCLUSIVE or MODIFIED) state. The former use the RDS request type, and the latter use RDEX. In either case, the IceT processor may need to displace a block from the L2 cache in order to make room for the newly fetched block. When a shared read request needs to displace a block that is in the EXCLUSIVE or MODIFIED state, the processor issues an RDSV (ReaD Shared with Victim writeback) request. Displacing reads to read-write data use the RDV (ReaD exclusive with Victim writeback) command.

Processing of reads with victim writebacks proceed pretty much as I've described for any other victimless read. But in addition to the process outlined before, we add a few operations beginning with **Step 4**

Step 4V

The L2 cache controller chooses a victim from among the four cache ways. If the victim block is in the EXCLUSIVE or MODIFIED state, the L2 changes the command request from RDS to RDSV or from RDEX to RDV as appropriate, and launches a victimization probe to the L1 cache for the victim address Y. (If the block is in the SHARED state, the L2 simply requests that the L1 invalidate block Y if it is present in the L1 Dcache.) The RDSV and RDV operations will inform the COH that this processor intends to write address Y back into memory.

Step 5V

The L1 probes its cache to see if the victim block Y is in its cache. If so, and the block is MODIFIED, the data is extracted and sent to the CSW interface. At the same time, the block is marked invalid in the L1.

Step 6V

If the L1 probe did not hit in the L1 Dcache, or it hit on EXCLUSIVE (unmodified) data, *and* if the state of the victim in the L2 was EXCLUSIVE, the processor sends a WBCANCEL (WriteBack CANCEL) request to the COH to terminate the writeback operation.

If however, the L1 probe hit on MODIFIED data, the L1 data is sent to the COH to complete the writeback.

Otherwise, data from the L2 lookup is sent to the COH.

Step 7V

Upon receiving the data, the COH writes the victim block to memory and completes the writeback transaction.

Simultaneous Accesses

Simultaneous is something of a misnomer here. Since commands arrive one-at-a-time at the CSW to COH interface, the COH unit acts as a synchronization and sequencing point for all transactions. However, it is possible for two or more processors to have an access outstanding to the same address at the same time.

The COH processes read miss requests in first-come/first-served order. Imagine that processor **A** sends a RDEX for address X to COHE, and that processor **B** sends a RDEX for the same address a few cycles later. COHE first passes **A**'s request through the shadow tags, registers **A**'s request in a table of outstanding read request called the ORC (Outstanding Read CAM). The table can be accessed as a CAM with the address as the key. The COH sends address X to the DRAM controller.

When COHE processes **B**'s request, it will find a hit in the ORC *and* a hit in the shadow tags suggesting that **A** is the current owner of the block. Rather than forwarding a probe request directly to **A**, the COH makes a new entry in the ORC for **B**'s request and marks it for later processing.

At some later time, the DRAM will respond with the read data for **A**'s original request. The COH will forward the data to **A** and scan the ORC for any transaction that was waiting for **A**'s request to complete.

The COH will find **B**'s request in the ORC. It will then forward a PRBWIN on **B**'s behalf to processor **A** where the transaction will be completed as any other shared data transfer.

Just as the COH must be handle memory reads in sequence, read operations that require a forwarded probe must be handled in sequence relative to earlier writes. The COH logs all writeback operations in the Write Back CAM (WBC) using the address as the key. The first read that hits on a WBC entry must be pended to the corresponding write operation. When the write operation is complete, the COH picks up where it left off with the read request and services it with either a DRAM reference, or in the case of some write operations, with a forwarded probe request.[6]

Block Transfers

Early in the design of Ice9 we decided that block DMA transfers through IO devices or through the communications controller should not displace or victimize data from L2 caches. In IceT we go one step further and ensure that block transfers don't displace the object data from either the L1 Dcache or the L2 cache in any processor.

Consider the case where the PCI express controller wishes to read the aligned 64 byte block starting at location X where bit 6 is zero.

Step 1R

In this case, the PCI express will send a BRD (Block ReaD) request to COHE.

Step 2R

COHE will perform the same lookup on address X as if this were a RDS operation. If no cache owns the data, the request will be satisfied by DRAM and proceed along a sequence similar to that for a simple load miss. Otherwise if the block is currently "owned" by processor B, COHE will send a PRBBRD (PRoBe Block ReaD) request to B.

Step 3R and Following

Processor **B** will handle the PRBBRD request as if it were a PRBSHR request, but it will leave the block in whatever state it began in. That is, the data will be forwarded to the PCI express widget, but the block will be retained in memory.

[6]The COH issues a forwarded read probe when a write completes only for block write operations, discussed below, and in reaction to a completed WRSTRANS. For more on the WRSTRANS sequence, see the IceT spec.

The SiCortex protocol also supports a Block WriTe (BWT) transaction to allow transfers from the outside world into a node's memory. In this case, the transaction flow is unlike any we've discussed so far.

Consider a BWT operation on address X that is currently in the EXCLUSIVE state in processor **A**. The operation originates in the PCI express widget.

Step 1W

The PCI express widget sends a BWT request to COHE.

Step 2W

COHE finds address X is currently owned in the EXCLUSIVE (or MODIFIED, the COH has no way of knowing) state in processor **A**. COHE forwards a PRBBWT to processor **A**.

Step 3W

Processor **A** performs a lookup of address X in its L1 and L2 caches.

Step 4W

If the block is not resident, **A** sends a BWTNOHIT to the PCI express controller and the transaction ends when the PCI express controller sends write data to the COH where it is passed on to the DRAM. Otherwise, processor **A** places a lock on the L1 line that could contain X and sends a BWTGO command to the PCI express controller.

Step 5W

The PCI express controller sends its write data to processor **A**.

Step 6W

Processor **A** accepts the data from the PCI express controller and writes it into the appropriate L1 or L2 cache block.

Step 7W

Processor **A** sends a BWTDONE command back to COHE.

Step 8W

On receipt of the BWTDONE command, the COHE cleans up any residual state from this command.

A BWT operation to a block X in the SHARED state in any cache will cause the COH to send a PRBINV (PRoBe INValidate) to all caches, forcing them to mark block X invalid. BWT operations to blocks in the SHARED state or blocks that are not currently cache resident are written by the COH directly to DRAM.

Summary: The Cache State Transition Diagrams

As I've said, this memo is not to be viewed as authoritative. But, the following diagrams have been helpful in understanding the cache state transition paths for various operations.

Figure 1 describes the state transitions that are caused by local processor activity. Figure 2 describes the state transitions that are caused by incoming probe requests caused by activity on other processors.

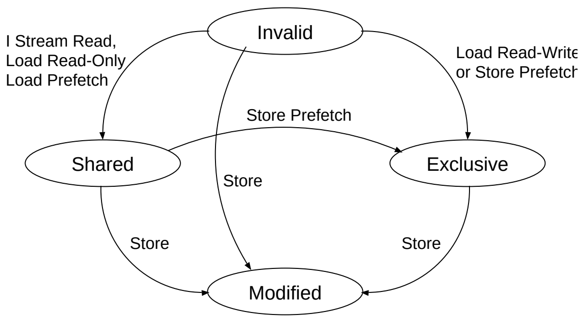

Fig. 1. Cache Transitions Caused by Local Processor Activity

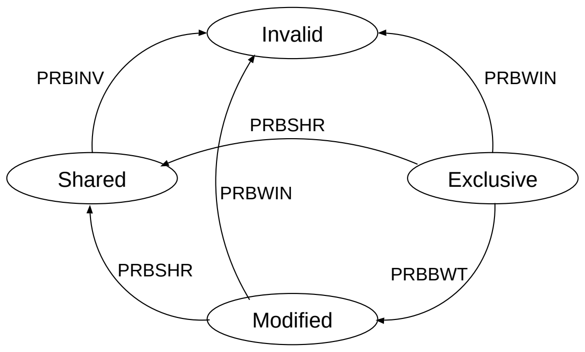

Fig. 2. Cache Transitions Caused by Probe Requests