Overview

BedRock is a family of directory- and invalidate-based cache coherence protocols that employ valid subsets of the common MOESIF coherence states. BedRock employs full tag duplication in the directory, and emphasizes design decisions that reduce controller complexity over the "performance at all costs" mentality. BedRock is designed for single-chip multicore processors and has been implemented in the BlackParrot RISC-V 64-bit Linux-capable multicore processor [3]. The BedRock protocols have been verified using CMurphi [4].

Readers unfamiliar with cache coherence in general or directory-based cache coherence specifically are strongly encouraged to read the excellent book titled "A Primer on Memory Consistency and Cache Coherence, Second Edition" by Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood [2] as background. This document largely employs their terminology and frames BedRock against the protocols described in their book.

The rest of this paper describes BedRock in detail, including the coherence system components, coherence state tracking and directory design, supported coherence states, and the coherence protocols at both a high level and with implementation details.

Coherence System Components

Local Cache Engine (LCE) / Cache Controller

The Local Cache Engine (LCE), or cache controller, is responsible for maintaining the coherence state of a core's private cache(s). In a typical system, a private cache is a Level 1 cache, such as a core's L1 Data or Instruction cache, or a core's private unified L2 cache. Accelerator or I/O devices (with or without caches) also have LCEs, regardless of whether the device participates in the coherence protocol. BedRock requires that the private cache hierarchy per core is inclusive.

Cache Coherence Engine (CCE) / Coherence Directory

The Cache Coherence Engine (CCE), or coherence directory, maintains cache coherence for a subset of the physical address space. Coherence is maintained at the cache block granularity. There are one or more CCEs in a BedRock coherence system. Each CCE operates independently from all other CCEs. A simple mapping of cache blocks to CCEs results in cache blocks being striped across the CCEs[1], e.g., Directory = Block Number % Number of CCEs. The coherence directory in the CCE stores full copies of the LCE cache tags and state that are managed by the CCE. BedRock's coherence directory is most similar to a shadow tag directory in organization.

[1] In general, the mapping is implementation specific. The first implementation of BedRock uses a simple striping of blocks across the CCE because of its low hardware cost and simplicity.

The rest of this document describes the protocol as if there is a single CCE. Since multiple CCEs manage distinct subsets of the address space and operate independently the only added complexity for multiple directories is the proper routing of messages from cache controller to directory and proper indexing of address into directory storage within the directory.

Coherence Networks

BedRock employs three logical coherence networks. These networks carry coherence requests, commands, and responses. Network implementation is system specific and independent of the protocol. All three networks may be totally ordered, point-to-point ordered, or unordered.

BlackParrot's BedRock implementation uses one physical network per logical network. Each network is wormhole routed, point-to-point ordered, and uses dimension ordered routing.

Request Network

The Request Network carries coherence requests from the cache controllers (LCEs) to the directory (CCE). A request may be either a Read request or Write request. Upgrade requests (write miss on a read-only block) are sent as standard Write requests to avoid races between the upgrade and a forwarded request message found in many protocols. Uncached requests are also supported[2].

Command Network

The Command Network carries coherence commands to the cache controllers (LCEs). Most commands are issued by the directory (CCE), but cache block data and its associated coherence state and address may also be sent from one cache controller (LCE) to another, at the direction of a previous command from the directory (CCE).

Response Network

The Response Network carries coherence responses from the cache controllers (LCEs) to the directory (CCE). Common responses include command acknowledgements and cache block data messages (e.g., writebacks).

Coherence Network Priority

The three coherence networks are ordered in priority, from highest to lowest as follows:

1. Response Network
2. Command Network
3. Request Network

Processing a message on a lower priority network may cause a message to be sent on a higher priority network, but not the other way around. It is also possible for a Command Network message to result in the sending of a second message on the Command Network, but this is strictly limited to a single message being sent, which is then either sunk or responded to with a single message on the Response Network, thereby preserving ordering. This ordering property

[2] In the near future, uncached access to cacheable memory space will be supported. L2/memory atomic operations will also be added in the future.

helps guarantee deadlock-free operation of the protocol, and is a common property for many types of networks. See Sections 8.2.3 and 9.3 in [2] for background on protocol deadlock.

Coherence System Diagram

Figure 1 shows how the various components in BedRock connect. For clarity, connection lines are shown for only a single LCE and CCE plus LCE to LCE messages. Each LCE (cache controller) sends messages on the Request and Response Networks to the CCEs (coherence directory), and sends and receives messages on outbound and inbound channels of the Command Network, respectively. CCEs receive messages on the Request and Response Networks and send messages on the Command Network. The blue lines show the LCE to LCE message path via the Command Network. LCE to LCE commands are only sent in response to an LCE receiving a specific command from the CCE. Only one command from an LCE may be generated from an appropriate command sent by the CCE, and this LCE to LCE command is sunk at the target LCE, possibly generating a single message on the Response Network back to the CCE.

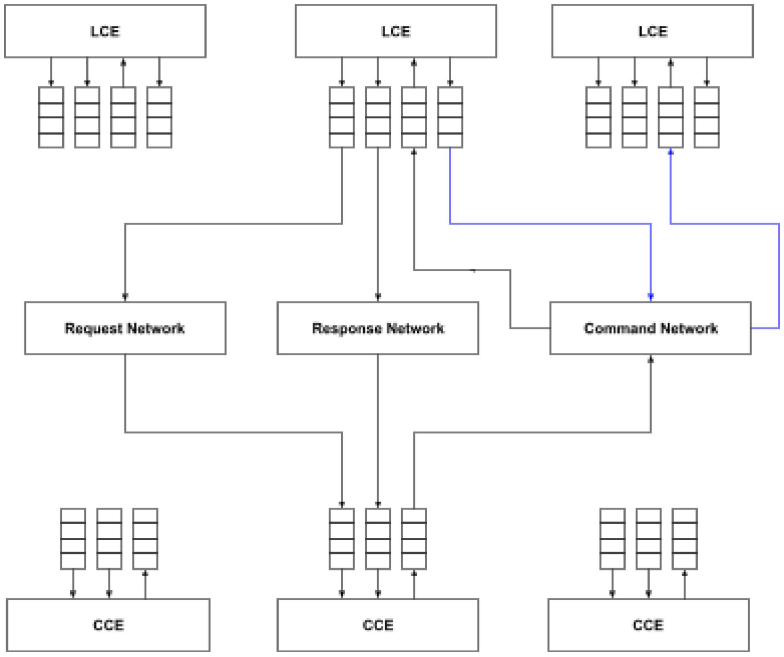

Figure 1. BedRock Coherence System

Coherence State Tracking

Cache Controller (LCE) - Tag Sets

Each LCE manages a single cache (or the lowest level cache of a private inclusive cache hierarchy at a core). The LCE tracks the coherence state for every cache block in the local cache using the concept of Tag Sets. Figure 2 depicts a Tag Set, which is the collection of coherence state and address tag for each way within a single cache set. The pair of coherence state and address tag for a single way is called a Tag Set Entry. There is one Tag Set per cache

set in the cache and one Tag Set Entry per way in each cache set. The cache and LCE must also manage replacement metadata, for example the LRU bits and LRU way to use when evicting and replacing a cache block on a cache-miss.

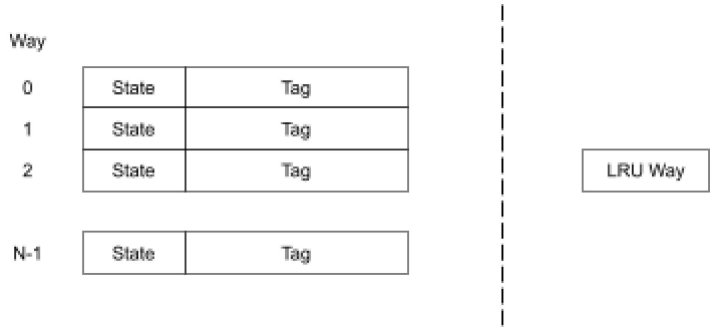

Figure 2. Tag Set and Metadata

The LCE stores what we call *shadow copies* of the Tag Sets. The *golden* or *true* Tag Sets are stored by the CCE. In general, the LCE may not modify the coherence state or tag of any Tag Set Entry unless being explicitly commanded to by the CCE. The sole exception to this is when a block is cached in the Exclusive (E) state and is stored to by the cache, resulting in the block changing to the Modified (M) state to indicate the block is now dirty. Any cache request (load, store, atomic) that requires permissions beyond those currently present results in the LCE issuing a coherence request on the Request Network to the CCE.

Coherence Directory (CCE) - Tag Sets and Way Groups

The CCE is organized such that it tracks every cache block in all of the LCEs. The CCE stores the complete Tag Sets from every LCE in the system using a collection of Directory Segments. Each segment stores the Tag Sets for some number of identically organized LCEs. There are one or more Directory Segments in the CCE, depending on how many different types of LCEs there are in the system. For example, a common system employs different cache organizations for the private instruction and data caches, each with an attached LCE. In this scenario, the CCE has two Directory Segments, one to track the instruction caches and a second to track the data caches. The CCE stores the *golden* or *true* copies of the Tag Sets and has full control over the coherence state for every cache block.

A Way Group is a collection of cache blocks (physical addresses) that are related and is used to order coherence transactions for related blocks. At one extreme, the CCE can use a single way group and totally order all coherence transactions. At the other extreme, the CCE can use one way group per cache block sized chunk of the physical memory space (although this would require MEM_SIZE_BYTES / CACHE_BLOCK_SIZE_BYTES) way groups and associated tracking state). In practice, the CCE employs one way group per cache set (assuming all LCE's are organized identically). A hash function is used to map a request's physical address to a way group, and all accesses that map to the same way group are serialized with one another. Requests to unique way groups are independent by definition and can proceed concurrently in

regards to the cache coherence protocol[3]. Each way group has an associated Pending Bit that tracks whether the way group already has an outstanding coherence transaction being processed. In order to maintain correctness in the coherence and memory system, a CCE only processes a new LCE Request if the pending bit is cleared for the way group associated with the request address.

In practice, the pending bit is implemented as a small counter. This counter is incremented when the CCE begins processing a new LCE Request and when the CCE sends a command to memory while processing the request. The counter is decremented when the CCE receives and processes responses from memory and the coherence acknowledgement from the LCE. The protocol makes no assumptions about the latency of messages between the LCEs and CCEs or the CCEs and memory, and a transaction is only complete when all memory commands have been completed and the LCE has responded with a coherence acknowledgement, which may arrive before or after the last memory response.

Coherence Protocol

Notes on Terminology

In this section we try to use the standard “cache controller” and “directory” terms to refer to the LCE and CCE, respectively, so that readers familiar with the descriptions in [1,2] can more readily understand the protocol.

Coherence States

The BedRock protocol fully supports any valid subset from the MOESIF family of coherence states[4]. The excellent definitions and terminology from [1,2] are used to describe the states. Every cache block exists in one of these states and can be described using four properties: validity, dirtiness, exclusivity, and ownership. The table defines the states in terms of the last three properties, where validity is determined if any of the three properties’ bits is set. The states described use a cache-centric view, that is they describe the state of the block from the view of the local cache controller.

A block is valid if its value is the most up-to-date value for that block in the system. A block is dirty if its value is the most up-to-date in the system and the value differs from the value of the block in memory. The opposite of dirty is clean. A block is exclusive if it is the only privately cached copy of the block in the system. The opposite of exclusive is non-exclusive or shared. A block is owned if the cache controller is responsible for responding to coherence requests for the block. In BedRock, an owned block may be transferred from one cache to another, at the direction of the directory controller.

---

[3] Any ordering required to satisfy the memory consistency model of the system must be enforced by software or the hardware issuing memory access requests (i.e., the LCE).

[4] The FSM CCE implements MESI, the ucode CCE has up to date ucode for EI, MSI, MESI, and MOESIF protocols as of 10/1/2020.

State	Properties			Permissions
	Dirty	Owned	Not Exclusive / Shared	
Invalid (I)	0	0	0	None
Shared (S)	0	0	1	R
Exclusive (E)	0	1	0	R/W
Modified (M)	1	1	0	R/W
Owned (O)	1	1	1	R
Forward (F)	0	1	1	R

Notes

- Owned is a read-only state where the block is dirty relative to memory. One controller will have the block in Owned and any other caches will have the block in S.
- Forward applies ownership to a Shared block. The block is up-to-date in memory. One controller will have the block in Forward and any other caches will have the block in S.

Atomic Transactions

Similar to some snooping-based protocols, BedRock employs Atomic Transactions. As described in Section 7.2.2 of [2], “The Atomic Transaction property ensures that no subsequent requests for a block will occur until after the current transaction completes, eliminating the need to handle requests from other cores while in one of these transient states.” In BedRock, this means that only one coherence transaction is processed at a time by the coherence directory (CCE) for a given cache block. Current implementations of the BedRock directory are still able to process transactions for different, independent cache blocks concurrently, for example overlapping memory access or waiting for a coherence ack to close a transaction with the processing of an independent transaction.

Since BedRock is a directory protocol and not a snooping protocol, it is possible for multiple coherence controllers to issue Read or Write Request messages for the same block at the same time without violating the Atomic Transaction property. However, only one of these requests will be processed at a time. A transaction is considered to start when the coherence directory begins processing the request message and ends when the coherence directory receives all response messages from memory and the coherence acknowledgement message from the requesting coherence controller. BedRock uses the Pending Bits to enforce the Atomic Transaction property while allowing concurrent processing of transactions to different blocks.

Protocol Messages

The following lists enumerate the messages carried on the three coherence networks. The specific messages required for the implemented protocol depends on which protocol, or specific valid subset of MOESIF states, is supported. Some messages may be optional, but usually helpful, to support in the cache controllers. It is recommended that cache controllers support all messages for all states, except those state and message pairs that are impossible in the target protocol(s). By design, BedRock relies on the directory to implement protocol correctness assuming that the cache controllers provide the correct response or take the correct action for all messages received.

Request Network

- Read Request (load-miss)
- Write Request (store-miss)

Modifiers

- Non-exclusive: hint that read request should be returned without exclusivity/ownership
- LRU way index: cache way to use for block fill, if required

Command Network

- Invalidate
    - set coherence state of the target cache block to Invalid
    - cache controller responds with an Invalidate Ack message
- Set State
    - set the coherence state of the target Tag Set Entry to the state specified in the message
    - cache controller sends no response
- Set Tag + Data
    - set the coherence state and tag of the target Tag Set Entry to the tag and state specified, fill cache block with provided data, and consider current miss resolved
    - cache controller responds with Coherence Ack message
- Set State + Wakeup
    - set the coherence state and tag of the target Tag Set Entry to the tag and state specified, and consider current miss resolved
    - cache controller responds with Coherence Ack message
- Writeback
    - cache controller responds with Writeback or Null Writeback message, depending on the current coherence state of the cache block
    - cache block does not change state
- Set State + Writeback
    - cache controller responds with Writeback or Null Writeback message, depending on the current coherence state of the cache block, and changes coherence state of the Tag Set Entry to state provided
- Transfer

    - cache controller sends Set Tag + Data command to target cache controller specified by Transfer command
- Set State + Transfer
    - cache controller sends Set Tag + Data command to target cache controller specified by Transfer command, and changes coherence state of the Tag Set Entry to state provided
    - e.g., transfer from M results in changing state to O in response to read request
- Set State + Transfer + Writeback
    - cache controller sends Set Tag + Data command to target cache controller specified by Transfer command, responds with Writeback or Null Writeback message, depending on the current coherence state of the cache block, and changes coherence state of the Tag Set Entry to state provided
    - e.g., transfer from M results in changing state to S in response to read request
    - e.g., transfer from M results in changing state to I in response to write request
- Sync
    - for init routine only
- Set Clear
    - clear entire cache set in LCE
    - not currently used by CCE, but may be useful for CCE directed cache clearing in future applications

Response Network

- Invalidate Ack
    - cache controller responds to directory acknowledging it has invalidated the block
- Coherence Ack
    - cache controller responds to directory acknowledging it has received Set Tag + Wakeup or Set Tag + Data message to complete its cache miss
- Writeback
    - cache controller responds with cache block data to directory after receiving a writeback command
- Null Writeback
    - cache controller responds to directory after receiving a writeback command, but sends no data because the block was clean at the cache controller
- Sync Ack
    - acknowledge Sync Cmd during initialization routine

Protocol Message Equivalency

The table below compares messages in BedRock to the standard messages in [2]. In BedRock, the writeback message (equivalent of Put in [2]) is initiated by the directory, not the cache controller.

BedRock Network	**BedRock Message**	**Closest Equivalent from [2]**
Request	Read Request	GetS
	Write Request	GetM, Upgrade
Command	Invalidate	Invalidate
	Set Tag + Data	Data from Dir or Owner
	Set State	Put (Downgrade S or F to I)
	Set State + Wakeup	Upgrade Response
	Writeback	Replacement / Put
	Set State + Writeback	Replacement / Put
	Transfer	Fwd-Get
	Set State + Transfer	Fwd-Get
	Set State + Transfer + Writeback	Fwd-Get + Replacement
Response	Invalidate Ack	Invalidate Ack
	Coherence Ack	
	Writeback	PutM, PutO
	Null Writeback	PutS, PutE, PutF

Next State Tables

This section shows the coherence state transition tables for both the cache controller and coherence directory for a number of valid subsets of the MOESIF states. Actions that do not require a coherence state change may not be shown (e.g., a load when the cache controller has the block in S or M state).

Cache Controller / LCE

For the cache controller, any action taken on a starting state not listed does not result in a state change at the cache controller (i.e., the action can be resolved without a coherence request).

MI

Action	Current State	Next State
Load	I	M
Store	I	M
Other Load	M	I
Other Store	M	I

MSI

Action	Current State	Next State
Load	I	S
Store	I, S	M
Other Load	M	S
Other Store	S, M	I

MESI

Action	Current State	Next State
Load	I	S, E
Store	I, S	M
Store (silent upgrade)	E	M
Other Load	E, M	S
Other Store	S, E, M	I

MESIF

Action	Current State	Next State
Load	I	S, E
Store	I, S, F	M
Store (silent upgrade)	E	M
Other Load	E, M	F
Other Store	S, E, M, F	I

MOSI

Action	Current State	Next State
Load	I	S, E
Store	I, S, O	M
Other Load	M	O
Other Store	S, O, M	I

MOSIF

Action	Current State	Next State
Load	I	S, F
Store	I, S, O, F	M
Other Load	M	O
Other Store	S, O, M, F	I

MOESI

Action	Current State	Next State
Load	I	S, E
Store	I, S, O	M
Store (silent upgrade)	E	M
Other Load	E	S
	M	O
Other Store	S, E, M, O	I

MOESIF

Action	Current State	Next State
Load	I	S, E
Store	I, S, O, or F	M
Store (silent upgrade)	E	M
Other Load	E	F
	M	O
Other Store	S, E, M, O, or F	I

Directory Controller / CCE

The directory controller tables show the current state of the directory and the new state recorded at both the directory and provided to the requesting cache controller. The Process column indicates the steps the directory must take to complete the request correctly, including invalidating the block in other cache controllers, reading the block from memory, modifying the state of the block in a cache controller, or transfering the block from one cache to another.

Note: Invalidate in the Process column means invalidate sharers (caches with block in S state), but not owner.

MI

Action	Current State (Dir/Owner)	Next State (Dir/Owner)	Next State (Requestor)	Process
Read	I	M	M	Read memory
	M	M	M	Transfer+State(I)
Write	I	M	M	Read memory
	M	M	M	Transfer+State(I)

MSI

Action	Current State (Dir/Owner)	Next State (Dir/Owner)	Next State (Requestor)	Process
Read	I, S	S	S	Read memory
	M	S	S	Transfer+Writeback+State(S)
Write	I	M	M	Read memory
	S	M	M	Invalidate+Read memory or Invalidate+ST-Wakeup
	M	M	M	Transfer+State(I)

MESI

Action	Current State (Dir/Owner)	Next State (Dir/Owner)	Next State (Requestor)	Process
Read	I	E	E	Read memory
	S	S	S	Read memory
	E	S	S	Transfer+Writeback+State(S)
	M	S	S	Transfer+Writeback+State(S)
Read (Non-Exclusive)	I	S	S	Read memory
	S	S	S	Read memory
	E	S	S	Transfer+Writeback+State(S)
	M	S	S	Transfer+Writeback+State(S)
Write	I	M	M	Read memory
	S	M	M	Invalidate+Read memory
	E, M	M	M	Transfer+State(I)

MESIF

Action	Current State (Dir/Owner)	Next State (Dir/Owner)	Next State (Requestor)	Process
Read	I	E	E	Read memory
	S	S	S	Read memory
	E	S	S	Transfer+Writeback+State(S)
	M	S	S	Transfer+Writeback+State(S)
	F	F	S	Transfer
Read (Non-Exclusive)	I	S	S	Read memory
	S	S	S	Read memory
	E	S	S	Transfer+Writeback+State(S)
	M	S	S	Transfer+Writeback+State(S)
	F	F	S	Transfer
Write	I	M	M	Read memory
	S	M	M	Invalidate+Read memory
	F	M	M	Invalidate+Transfer+State(I)
	E, M	M	M	Transfer+State(I)

MOSI

Action	Current State (Dir/Owner)	Next State (Dir/Owner)	Next State (Requestor)	Process
Read	I	S	S	Read memory
	S	S	S	Read memory
	O	O	S	Transfer
	M	O	S	Transfer+State(O)
Write	I	M	M	Read memory
	S	M	M	Invalidate+Read memory
	O	M	M	Invalidate+Transfer+State(I)
	M	M	M	Transfer+State(I)

MOSIF

Action	Current State (Dir/Owner)	Next State (Dir/Owner)	Next State (Requestor)	Process
Read	I	F	F	Read memory
	S	S	S	Read memory
	O	O	S	Transfer
	M	O	S	Transfer+State(O)
	F	F	S	Transfer
Write	I	M	M	Read memory
	S	M	M	Invalidate+Read memory
	O, F	M	M	Invalidate+Transfer+State(I)
	M	M	M	Transfer+State(I)

MOESI

Action	Current State (Dir/Owner)	Next State (Dir/Owner)	Next State (Requestor)	Process
Read	I	E	E	Read memory
	S	S	S	Read memory
	E	S	S	Transfer+Writeback+State(S)
	M	O	S	Transfer+State(O)
	O	O	S	Transfer
Read (Non-Exclusive)	I	S	S	Read memory
	S	S	S	Read memory
	E	S	S	Transfer+Writeback+State(S)
	M	O	S	Transfer+State(O)
	O	O	S	Transfer
Write	I	M	M	Read memory
	S	M	M	Invalidate+Read memory
	O	M	M	Invalidate+Transfer+State(I)
	E, M	M	M	Transfer+State(I)

MOESIF

Action	Current State (Dir/Owner)	Next State (Dir/Owner)	Next State (Requestor)	Process
Read	I	E	E	Read memory
	S	S	S	Read memory
	E	F	S	Transfer+Writeback+State(F)
	M	O	S	Transfer+State(O)
	O	O	S	Transfer
	F	F	S	Transfer
Read (Non-Exclusive)	I	S	S	Read memory
	S	S	S	Read memory
	E	F	S	Transfer+Writeback+State(F)
	M	O	S	Transfer+State(O)
	O	O	S	Transfer
	F	F	S	Transfer
Write	I	M	M	Read memory
	S	M	M	Invalidate+Read memory
	O, F	M	M	Invalidate+Transfer+State(I)
	E, M	M	M	Transfer+State(I)

Directory / CCE Request Processing Flow

As mentioned above, the directory processes one request at a time. This allows BedRock to maintain Atomic Transactions and eliminate transient states (or their external exposure) from the protocol. The steps below provide the most straightforward processing algorithm for the directory. After this initial discussion it will be shown how to optimize the implementation to increase performance, such as not stalling the directory while it waits for a memory read to return, which is potentially a long latency operation.

Basic Processing Flow

The flow below describes how the directory processes a request. It assumes a MOESIF protocol, but is easily adjusted (simplified) for any valid subset of MOESIF as the protocol.

1. Read or Write request arrives
    a. Read Request may be non-exclusive
    b. Request specifies way of block to use for replacement / fill (LRU way)
        i. directory will lookup current state of the block to determine if it needs to be written back to memory (replacement)
2. Check Pending Bit, read Directory, GAD, determine next coherence state of block and coherence state of LRU way block
3. If request can be satisfied as an upgrade
    a. replacement of LRU block not needed because the block is not being evicted
    b. invalidate block from all other sharer LCEs (controllers in S)
    c. if block is in O or F and owner is not requestor
        i. transfer block from owner to requestor, invalidate owner
    d. else respond with Set State and Wakeup giving read-write permissions (and ownership/exclusivity) to requestor
4. Replace LRU block if directory read indicates LRU block is valid
    a. for block in S or F, invalidate block
    b. for block in E, M, or O, invalidate and writeback block
        i. directory issues writeback to memory
5. Invalidate sharers (cache controllers with block in S) if write request
6. Transfer block from owner or read block from memory
    a. if no owner exists for the block, read from memory and forward to requestor when memory responds
    b. if owner exists and write request, issue transfer + set state command to owner
        i. owner will send set tag + data to requestor
        ii. owner will set state to invalid
    c. if owner exists in O or F and read request, issue transfer command to owner
        i. owner will send set tag + data to requestor
        ii. owner remains in current state
    d. if owner exists in E (at directory, may be in E or M at controller) and read request, issue transfer + set state + writeback command to owner

        i. owner will send set tag + data to requestor
        ii. owner sends writeback (if in M) or null writeback (if in E) response to directory
        iii. owner will set state to F
        iv. directory waits for writeback response and forwards to memory if not null
    e. if owner exists in M and read request, issue transfer + set state command to owner
        i. owner will send set tag + data to requestor
        ii. owner will set state to O
7. Directory waits for responses, which may arrive in any order
    a. requestor responds with coherence ack to indicate transaction is complete from its point of view
    b. memory responds with writeback done response if writeback occurred during replacement
    c. memory responds with writeback done response if writeback occurred during transfer

Replacements

A miss request from a cache controller may require the LRU block indicated to be replaced. When the directory processes a request it reads the coherence directory, including reading the specific entry from the requesting controller indicated by the LRU way specified in the request message. The directory determines the state of the LRU block and performs a replacement if the block is cached in a valid state. Clean, read only blocks can either be invalidated or left alone, as they will be overwritten by the cache block fill from the request. Blocks in a potentially dirty state are invalidated and written back to memory. Replacements do not change the coherence permissions of the target block in any other cache. The table below shows the current state of the block being replaced (LRU block) as recorded in the directory and at the cache controller, the message the directory sends, and the response received by the directory. The next state of the block is always Invalid at the coherence controller. The next state of the block in the system depends on the initial state. Blocks in E or M will be invalid, blocks in S, O, or F may still be cached in S at other controllers, and therefore may be in Invalid or Shared.

Current State of LRU block recorded in Dir	Current State of LRU block in Coherence Controller	Message	Response
S	S	*Invalidate	InvAck
E	E	Set State + Writeback	NullWB
E	M	Set State + Writeback	WB
M	M	Set State + Writeback	WB
O	O	Set State + Writeback	WB
F	F	*Invalidate	InvAck

Note: the * indicates the invalidate is optional

By default, replacements are only performed when a cache controller issues a cache miss request and the directory discovers a replacement is required to satisfy the request. In the future, an external event could cause the directory to revoke permissions for a cache block, which can easily be handled by sending a writeback + set tag command or invalidate command to the appropriate cache controllers.

Processing Flow Optimizations

As mentioned above, the processing flow presented is not necessarily performance optimal. For example, the directory stalls waiting for memory to respond, which is a potentially long latency operation. In order to improve performance, the directory relies on the pending bits. The pending bits can be viewed as gatekeepers for the cache blocks. The pending bit is set as long as there is an active coherence transaction for the set of cache blocks associated with it.

Uncached Accesses

Globally uncached, locally uncached

BedRock supports uncached access to memory through special Request and Command messages. A coherence controller can issue an uncached read or write request to the directory, which is then forwarded to memory. Memory responds to the directory with either the requested load data, up to 64-bits in size, or a message indicating the uncached store was committed to memory. For loads, the directory forwards the data to the requesting coherence controller / cache using an uncached data command. For stores, the directory sends an uncached store done command to the requesting cache to inform it that memory has committed the store.

Globally cached, locally uncached

In some cases, coherence controllers / caches may request uncached access to a block that is globally cached, i.e., a block that is in cached address space and may be actively cached in other caches. In this case, the directory is read to determine if the block can be sourced from another cache or needs to be read from memory. The CCE invalidates the block, if found, from all caches, writes back the block to memory if it is dirty, and then provides the requested data to the LCE.

References

[1] Daniel J. Sorin, Mark D. Hill, and David A. Wood. “A Primer on Memory Consistency and Cache Coherence". In: Synthesis Lectures on Computer Architecture 6.3 (2011), pp. 1-212. doi: 10.2200/s00346ed1v01y201104cac016. url: https://doi.org/10.2200/s00346ed1v01y201104cac016.

[2] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. “A Primer on Memory Consistency and Cache Coherence, Second Edition". In: Synthesis Lectures on Computer Architecture 15.1 (2020), pp. 1-294. doi: 10.2200/S00962ED2V01Y201910CAC049. eprint: https://doi.org/10.2200/S00962ED2V01Y201910CAC049. url: https://doi.org/10.2200/S00962ED2V01Y201910CAC049.

[3] BlackParrot. https://github.com/black-parrot/black-parrot

[4] CMurphi. https://bitbucket.org/mclab/cmurphi/src/master/