The BedRock Cache Coherence Protocol and System

Mark Wyse

Updated: February 23, 2022

Contents

1 Introduction **4**

2 BedRock Cache Coherence System Components **5**
- 2.1 Canonical BedRock System . . . 5
- 2.2 BedRock Coherence Network . . . 5
  - 2.2.1 Request Network . . . 6
  - 2.2.2 Command Network . . . 6
  - 2.2.3 Fill Network . . . 6
  - 2.2.4 Response Network . . . 6
  - 2.2.5 Network Priority . . . 7
- 2.3 Cache Controller - Local Cache Engine (LCE) . . . 7
- 2.4 Coherence Directory - Cache Coherence Engine (CCE) . . . 8
- 2.5 BedRock System Assumptions . . . 9

3 BedRock Cache Coherence Protocol **10**
- 3.1 Address Space Properties . . . 10
- 3.2 Coherence Protocol States . . . 10
- 3.3 Protocol Messages . . . 11
  - 3.3.1 Request Network . . . 11
  - 3.3.2 Command Network . . . 13
  - 3.3.3 Fill Network . . . 13
  - 3.3.4 Response Network . . . 14
- 3.4 Coherence Transactions and Tracking Coherence State . . . 14
  - 3.4.1 Coherence Transaction . . . 14
  - 3.4.2 Cache Controller . . . 14
  - 3.4.3 Coherence Directory . . . 15
  - 3.4.4 Tracking State - Tag Sets . . . 15
  - 3.4.5 Ordering Transactions - Way Groups . . . 15
- 3.5 Protocol Assumptions . . . 16
- 3.6 Coherence Protocol Tables . . . 17
  - 3.6.1 Cache Controller Protocol Table . . . 17
  - 3.6.2 Coherence Directory Protocol Table . . . 17
- 3.7 Coherence State Transitions . . . 21
  - 3.7.1 Cache Controller State Transitions . . . 21
  - 3.7.2 Coherence Directory State Transitions . . . 21
- 3.8 Coherence Directory Processing Flow . . . 22

3.9 Coherence Protocol Comparison . . . 23
3.9.1 Transient States . . . 23
3.9.2 Protocol Messages . . . 23
3.9.3 Coherence Networks . . . 25

4 BedRock Cache Coherence Protocol Verification 26

5 Uncached and Atomic Accesses in BedRock 27
5.1 Uncacheable Accesses . . . 27
5.2 Atomic Read-Modify-Write Operations . . . 27

A BedRock Cache Coherence Protocol Tables 29
A.1 Cache Controller Protocol Tables . . . 29
A.2 Coherence Directory Protocol Tables . . . 33

B BedRock Cache Coherence State Transition Tables 39
B.1 Cache Controller State Transition Tables . . . 39
B.2 Coherence Directory State Transition Tables . . . 41

References 43

List of Figures

1 Canonical BedRock Coherence System Organization. *The Last-Level Cache is an optional, memory-side cache that does not participate in the cache coherence protocol. 5
2 BedRock Coherence Networks . . . 6
3 Canonical BedRock Coherence Directory request processing flow . . . 8
4 Canonical Address Space Layout and Cacheability Properties . . . 10
5 Tag Set . . . 15
6 Way Group . . . 16
7 Canonical BedRock Way Groups . . . 16
8 Generic Coherence Directory Request Processing Flow . . . 22

List of Tables

1 BedRock Coherence State Properties . . . 11
2 BedRock Request Network Messages . . . 11
3 BedRock Command Network Messages . . . 12
4 BedRock Fill Network Messages . . . 13
5 BedRock Response Network Messages . . . 14
6 BedRock Cache Controller Protocol Table - MOESIF . . . 19
7 BedRock Coherence Directory Protocol Table - MOESIF . . . 20
8 BedRock Cache Controller Next State Table - MOESIF . . . 21
9 BedRock Coherence Directory Next State Table - MOESIF . . . 22
10 BedRock Protocol Message Equivalency . . . 24
11 CMurphi verification time and speedup - MESI protocols . . . 26
12 BedRock Cache Controller Protocol Table - MI . . . 29
13 BedRock Cache Controller Protocol Table - MSI . . . 29

14 BedRock Cache Controller Protocol Table - MESI 30
15 BedRock Cache Controller Protocol Table - MESIF 30
16 BedRock Cache Controller Protocol Table - MOSI 31
17 BedRock Cache Controller Protocol Table - MOSIF 31
18 BedRock Cache Controller Protocol Table - MOESI 32
19 BedRock Coherence Directory Protocol Table - MI 33
20 BedRock Coherence Directory Protocol Table - MSI 33
21 BedRock Coherence Directory Protocol Table - MESI 34
22 BedRock Coherence Directory Protocol Table - MESIF 35
23 BedRock Coherence Directory Protocol Table - MOSI 36
24 BedRock Coherence Directory Protocol Table - MOSIF 37
25 BedRock Coherence Directory Protocol Table - MOESI 38
26 BedRock Cache Controller Next State Table - MI 39
27 BedRock Cache Controller Next State Table - MSI 39
28 BedRock Cache Controller Next State Table - MESI 39
29 BedRock Cache Controller Next State Table - MESIF 39
30 BedRock Cache Controller Next State Table - MOSI 40
31 BedRock Cache Controller Next State Table - MOSIF 40
32 BedRock Cache Controller Next State Table - MOESI 40
33 BedRock Coherence Directory Next State Table - MI 41
34 BedRock Coherence Directory Next State Table - MSI 41
35 BedRock Coherence Directory Next State Table - MESI 41
36 BedRock Coherence Directory Next State Table - MESIF 41
37 BedRock Coherence Directory Next State Table - MOSI 42
38 BedRock Coherence Directory Next State Table - MOSIF 42
39 BedRock Coherence Directory Next State Table - MOESI 42

1 Introduction

The BedRock Cache Coherence System and Protocol define both a family of cache coherence protocols and the required coherence system components that implement a protocol. The BedRock system is constructed from three components: cache controllers (Local Cache Engines), coherence directories (Cache Coherence Engines), and the coherence network. The BedRock protocols are directory-based invalidate protocols using the standard MOESIF coherence states. Protocol variants are defined for the MI, MSI, MESI, MOSI, MOESI, MESIF, or MOESIF state subsets. BedRock relies on a duplicate tag, fully inclusive, standalone directory to precisely track the coherence state of every block cached within the coherence system. The coherence directory is the point of serialization for all coherence transactions, and coherence is enforced using the *Single-Writer, Multiple-Reader (SWMR) Invariant* and *Data-Value Invariant*.

The canonical BedRock system and protocol presented in this paper are well-suited for small to medium size in-order multicore processors. An initial implementation of BedRock exists within the BlackParrot 64-bit RISC-V multicore processor [PGW+20]. This implementation has proved invaluable for protocol and system exploration and refinement, and is fully open-source and available at `https://github.com/black-parrot/black-parrot`. Although the BedRock system and protocol has certainly been influenced by the design needs of BlackParrot, this document presents the protocol agnostic to system implementation.

Throughout this paper, we assume readers are familiar with the basics of cache coherence and recommend Nagarajan et al.'s overview [NSH+20] as background reading for those unfamiliar with the topic. Specifically, Chapters 2 and 8 of Nagarajan et al.'s work covers the basics of cache coherence and directory-based coherence protocols. As much as possible, we adopt and use terminology from their work, which is consistent with the majority of prior cache coherence literature, in an attempt to make this document as approachable as possible.

The rest of this paper is organized as follows. Section 2 describes the BedRock coherence system components. Section 3 describes the family of BedRock cache coherence protocols, including both high-level overviews and detailed tabular specifications. Section 4 describes verification of the MESI variant of BedRock using the CMurphi [PIM+04] model checker software. Section 5 discusses how uncached and atomic accesses can be incorporated into BedRock. Appendices A and B provide additional coherence protocol and next state tables, respectively, for protocols using subsets of the MOESIF coherence states.

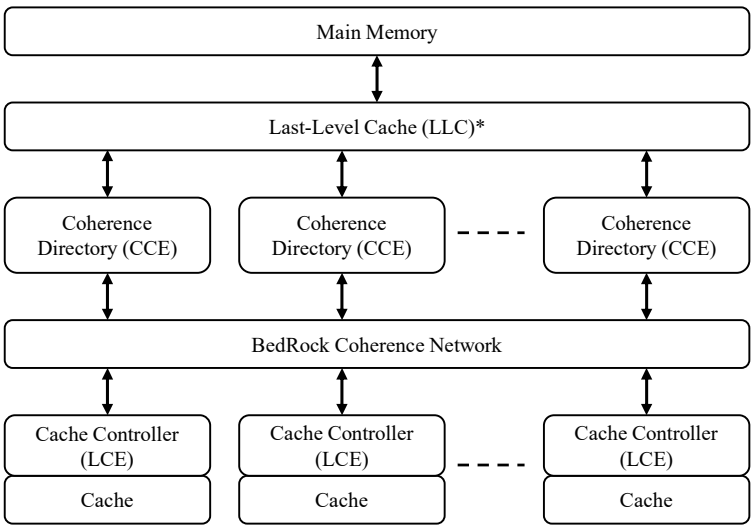

Figure 1: Canonical BedRock Coherence System Organization. *The Last-Level Cache is an optional, memory-side cache that does not participate in the cache coherence protocol.

2 BedRock Cache Coherence System Components

BedRock defines both a coherence protocol and the coherence system components that implement the protocol. This section describes the functionality of the cache controllers, coherence directories, and coherence network required to implement a BedRock coherence protocol. Section 3 describes the BedRock coherence protocol in detail. The discussion of BedRock begins with an overview of a canonical BedRock system, followed by descriptions of each of the coherence system components.

2.1 Canonical BedRock System

Figure 1 depicts a canonical BedRock coherence system. Each cache controller (LCE) manages a single cache. Each coherence directory manages a disjoint subset of the physical address space and contains a slice of the coherence directory storage associated with that subset. There may be one or many coherence directories (CCE), and two or more cache controllers. The controllers and directories are connected via the BedRock coherence network. The coherence directories also connect to main memory, with an optional memory-side, non-inclusive Last-Level Cache (LLC) between the directories and the main memory. The LLC does not participate in the cache coherence protocol, is logically considered to be part of main memory, and acts as a memory bandwidth amplifier for the lower-level caches. BedRock places no constraints on the organization of the LLC, but its implementation must provide a block-based access interface consistent with the cache block size of the BedRock system.

2.2 BedRock Coherence Network

The BedRock Coherence Network carries coherence protocol messages between the cache controllers and coherence directories. The BedRock coherence protocol comprises four distinct coherence networks to carry Request, Command, Fill, and Response messages. Figure 2 depicts the BedRock coherence networks and their connections to the cache controllers and coherence directories. The implementation of these networks is system specific, and independent of the BedRock

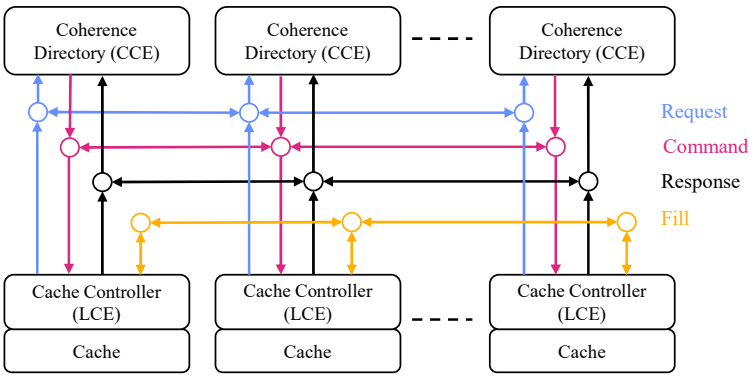

Figure 2: BedRock Coherence Networks

coherence protocol. The networks must only guarantee that messages are delivered error-free and each network operates independently. These networks may be implemented with physical networks or virtual networks carried by a single physical network. Each network may be totally ordered, point-to-point ordered, or unordered.

2.2.1 Request Network

The Request network carries messages from the cache controller to the coherence directory. Coherence requests are initiated when a cache miss occurs. Cache misses are the result of insufficient permissions to complete the cache operation. This includes both attempting to write to a read-only block and a read or write to a block that is not cached (i.e., no permissions exist for the block). The Request network is allowed to fill up and block cache controllers from issuing new requests. New requests will eventually send as the coherence directory drains and processes existing requests from the network. The network has no ordering constraints and requests are serialized by the network, eventually arriving as a single request stream at the directory.

2.2.2 Command Network

The Command network carries coherence commands from the coherence directory to the cache controllers. The Command network has no ordering constraints and only requires that commands are processed in a timely manner after arriving at the cache controller. A cache controller may not delay processing a command in order to send a new coherence request.

2.2.3 Fill Network

The Fill network carries cache to cache data transfers between the cache controllers. The Fill network has no ordering constraints and only requires that messages are processed in a timely manner after arriving at the cache controller. A cache controller may not delay processing a fill message in order to send a new coherence request.

2.2.4 Response Network

The Response network carries messages from the cache controller to the coherence directory in response to commands issued by the directory or forwarded from another cache controller. Responses include invalidation acknowledgements, writeback responses with or without data, and coherence transaction acknowledgements. The Response network has no ordering constraints. Each

response message has a single corresponding command that caused it to send. The coherence directory must process responses in a timely manner to prevent deadlock in the coherence system. The directory must also prioritize processing responses over processing new requests or issuing additional commands.

2.2.5 Network Priority

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

1. Response
2. Fill
3. Command
4. Request

When a cache controller or directory receives a message it may only cause a message of higher priority to send. Requests may cause the directory to send Commands, and Commands cause the cache controllers to send Fills or Responses. Fills cause the cache controllers to send Responses.

The cache controller and coherence directory must favor processing higher priority messages over lower priority messages to avoid deadlocking the protocol. Enforcing a priority ordering of the coherence networks helps guarantee deadlock-free operation of the protocol and is commonly used by many other protocols. We refer readers to Sections 8.2.3 and 9.3 in [NSH+20] for more information on deadlock avoidance.

2.3 Cache Controller - Local Cache Engine (LCE)

The cache controller in BedRock is called a Local Cache Engine (LCE) and manages coherence transactions for a single cache. The associated cache may be a private or shared cache, but is assumed to be a write-back cache that is inclusive of any lower-level caches in its hierarchy. The cache controller interfaces with its associated cache and with the BedRock coherence network. It may be tightly integrated into the cache pipeline or it may be more loosely coupled and interact with the cache over a well-defined interface that allows the controller to read and write cache block metadata and data.

Each cache controller manages coherence transactions for its associated cache. It issues new requests when a cache miss occurs and responds to coherence commands that arrive on the Command network. The associated cache is only allowed to access a block using the block's current permissions and may not change the permissions on a block unless directed by the coherence directory. Any operation that requires a change in permissions results in the controller issuing a new request. This includes cache block invalidations, which are detected and initiated by the coherence directory while processing coherence requests.

The cache controller must respond in a timely manner to coherence commands. Each coherence command typically generates a single response message. Cache to cache transfer commands may generate one or both of a single fill message to another controller and a single response to the directory, depending on the specific command. The cache controller must not stall command or fill message processing in order to issue a new coherence request. The BedRock system and protocol allow the request network to fill and block new requests from sending, but the command and fill networks must be processed as they arrive independent of the request network status.

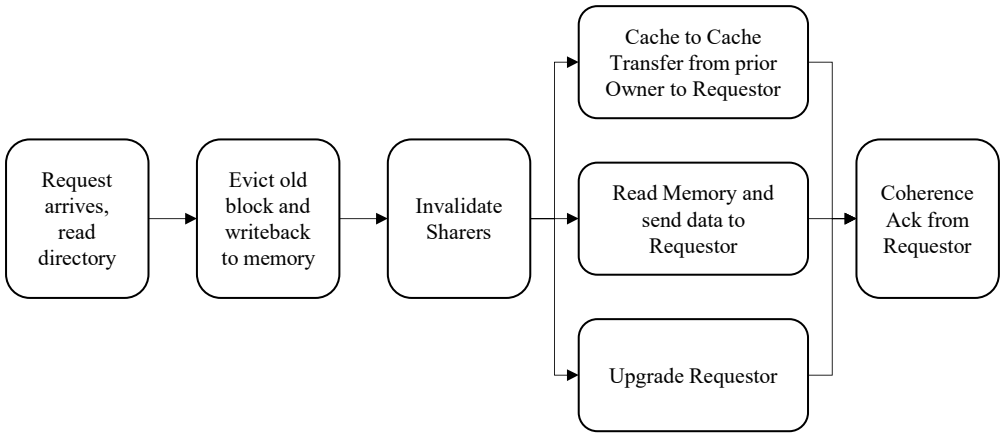

Figure 3: Canonical BedRock Coherence Directory request processing flow

2.4 Coherence Directory - Cache Coherence Engine (CCE)

The BedRock coherence directory is called the Cache Coherence Engine (CCE) and is responsible for maintaining coherence for a disjoint subset of the physical address space. A BedRock system may have one or more coherence directories. If multiple coherence directories exist, management of the address space is divided evenly among all coherence engines with the physical address space striped across directories at the cache set granularity. All cache blocks that map to the same cache set in a cache controller are managed by exactly one coherence directory. The canonical BedRock coherence directory is a fully inclusive, standalone, duplicate tag directory. All state transitions within the coherence protocol, including the eviction and replacement of cache blocks at the cache controllers, are controlled by the directory. The coherence directory keeps every cache block at every cache controller in a stable state at all times, and transient behavior is contained within the processing flow of the coherence directory.

The coherence directory must process response messages in a timely manner to prevent deadlock in the coherence protocol. The directory may stall additional requests and apply back-pressure on the request network while it processes the current request. The directory is also able to issue memory commands to fetch cache blocks or perform writebacks. The directory must either process memory responses as they arrive or provide sufficient buffering between the memory command and response channels to avoid stalling when issuing memory commands. A canonical system that processes requests in-order, as described below, must either block and wait for a memory response after each memory command or provide buffering for two memory command / response pairs per directory. The two memory operations that may be required for each coherence request are for cache block eviction writeback and either an additional writeback or a memory fetch.

The coherence directory processes coherence requests as they arrive. Each request results in one or more coherence commands being sent to the cache controllers. Request processing concludes when a coherence acknowledgment message is received at the directory from the controller that initiated the request. Figure 3 depicts the canonical, high-level request processing flow for the coherence directory. Requests arrive and the directory is read. If an eviction is required to make room for the newly request block and writeback is issued and then forwarded to memory if the cache responds with dirty cache block data. Next, any other caches with the block in the shared state are invalidated, if required for the specific request type. Then, the directory either initiates a memory read for the block, commands a cache to cache transfer (with a possible writeback to update memory), or responds with upgraded permissions if the requesting cache already has a

copy of the target block. Finally, the directory waits for a coherence acknowledgment message to complete the transaction.

A simple directory implementation may execute the processing flow serially, stalling to wait for responses for every cache controller or memory command. Optimized implementations, including the one in BlackParrot, may introduce concurrency between independent requests and even at selected places within a single request's processing. Logically, all requests are serialized relative to one another at the directory.

Any practical system must also be capable of processing uncacheable and atomic accesses to both cacheable and uncacheable memory. However, how these requests are handled and whether coherence is enforced for them is implementation specific. In general, atomic operations to cacheable memory should occur coherently while uncacheable operations to cacheable memory may or may not be coherent, depending on their use within the system. Section 5 discusses one approach to keeping both uncacheable and atomic accesses to cacheable memory coherent within the BedRock protocol and system.

2.5 BedRock System Assumptions

The following assumptions about the BedRock coherence system are made and used, unless explicitly stated otherwise, for the remainder of this document.

1. The coherence networks require no specific ordering properties, and may be completely unordered.
2. The coherence networks guarantee that messages are delivered error-free.
3. Each coherence network operates independently of the other networks.
4. Each cache controller manages a single cache that is inclusive of all lower-level caches in its hierarchy, if any exist.
5. All caches participating in coherence have the same organization (number of sets, associativity, and cache block size)
6. Each cache block is managed by a single coherence directory and all cache blocks that map to the same cache set are managed by the same coherence directory.
7. The coherence directory is a fully inclusive, standalone, duplicate tag directory.

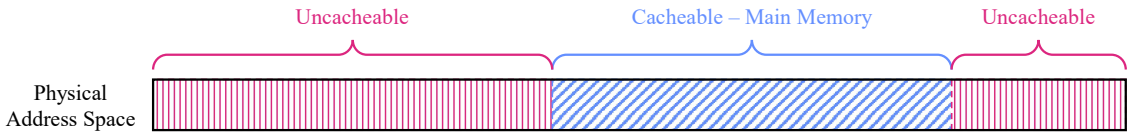

Figure 4: Canonical Address Space Layout and Cacheability Properties

3 BedRock Cache Coherence Protocol

BedRock's cache coherence protocol is a directory-based invalidate protocol featuring the common MOESIF coherence protocol states. BedRock is designed using a full-duplicate tag directory organization. The coherence protocol functions similarly to a standard directory protocol [NSH+20], however the coherence directory has complete control over all coherence state transitions in the protocol. In BedRock, the cache controllers may only use a cache block with its current permissions. Any change to permissions, including invalidation, must be requested from and directed by the coherence directory. BedRock's other major difference from canonical directory protocols is its use of four coherence networks, including a dedicated network for cache to cache data transfers.

The remainder of this section provides a deep dive into the operation of the BedRock cache coherence protocols, focusing on the fully-featured MOESIF variant. Section 3.1 defines the properties of the address space, and Section 3.2 examines the MOESIF cache coherence states and specifies the properties and permissions of each state. Section 3.3 describes the BedRock coherence protocol messages carried on each of the coherence networks. Section 3.4 defines the concept of a coherence transaction and explains how coherence state and transactions are tracked within BedRock. The assumptions imposed by the BedRock protocol are described in Section 3.5, followed by a presentation of BedRock's full coherence protocol tables for the MOESIF protocol in Section 3.6. Section 3.7 describes the possible coherence state transitions at both the cache controller and coherence directory for the MOESIF protocol. Section 3.8 describes the generic request processing flow of the coherence directory, and Section 3.9 provides a brief comparison of BedRock to a traditional directory-based coherence protocol.

3.1 Address Space Properties

BedRock defines a coherence protocol that is enforced for the cacheable region of the physical address space. Figure 4 depicts the canonical address space, with a single cacheable region backed by Main Memory (DRAM) and multiple uncacheable regions. Typically, the cacheable region of physical memory consists of the system's installed DRAM address space (or a subset thereof). Without loss of generality, the rest of this paper assumes this system model and that cacheable memory accesses are allowed only to cacheable memory (DRAM). Extending the cacheable address space to cover a differently sized or even multiple regions only requires the coherence directory to have knowledge about these ranges (or to trust the cache controllers to only issue cacheable access requests for cacheable memory). Uncacheable accesses are allowed to any physical address, and the handling of uncacheable accesses to cacheable memory is implementation specific. We discuss how BedRock is adapted to handle uncacheable accesses in Section 5.

3.2 Coherence Protocol States

Every cache block in the BedRock protocol exists in a stable coherence state: *Invalid (I)*, *Shared (S)*, *Exclusive (E)*, *Modified (M)*, *Owned (O)*, or *Forward(F)*. Adopting the terminology of [NSH+20], each coherence state is described using four well-defined properties: **validity**, **dirtiness**,

State	Name	Valid	Dirty	Owned	Not-Excl	RW	Encoding
I	Invalid	✗	✗	✗	✗	–	000
S	Shared	✓	✗	✗	✓	R	001
E	Exclusive	✓	✗	✓	✗	RW	010
F	Forward	✓	✗	✓	✓	R	011
M	Modified	✓	✓	✓	✗	RW	110
O	Owned	✓	✓	✓	✓	R	111

Table 1: BedRock Coherence State Properties

Message	Abbreviation	Description
Read Miss	ReqRd	Request cache block after a load miss
Read Miss (Non-Exclusive)	ReqRd (Non-Excl)	Request cache block after a load miss with hint to not provide block in E state
Write Miss	ReqWr	Request cache block after a store miss

Table 2: BedRock Request Network Messages

exclusivity, and **ownership**. Table 1 summarizes the mapping of properties to coherence states for the BedRock coherence protocol with exclusivity encoded as its negation (not-exclusive indicates the block may be shared and cached in one or more caches) and validity as the logical OR of the remaining three properties. A ✗ indicates the property is false, ✓ indicates the property is true. The **RW** column indicates if a block is read-only (**R**) or read-write (**RW**). A cache block is writable if the not-exclusive property is false (i.e., a single cache has ownership and write permissions) and the state is Valid. The **Encoding** column is a direct, three-bit encoding of the coherence state properties {*dirty, owned, not-exclusive*}, usable by hardware implementations.

3.3 Protocol Messages

As described in Section 2.2, the BedRock coherence protocol relies on the Request, Command, and Response networks to carry coherence protocol messages between the cache controllers and coherence directories. This section describes the messages carried on each of the coherence networks in detail. We present the messages for each network in a separate table. Each table lists the message name, its abbreviation as used throughout the remainder of this document, and a description of the message's functionality.

3.3.1 Request Network

Table 2 lists the Request Network message types that a cache controller may send to the coherence directory. Read requests are issued when the cache encounters a cache miss on a load operation, and write requests are issued when the cache encounters a cache miss on a store operation. Read requests may encode an optional Non-Exclusive hint to inform the coherence directory that there is no benefit in providing the cache block in the Exclusive coherence state instead of the

Message	Abbreviation	Description
Invalidate	Inv	Invalidate cache block specified by address
Data	DATA	Provide data and coherence state for block specified by address and wake up cache to complete request
Set State & Wakeup	STW	Modify state of cache block specified by address and wake up cache to complete request
Writeback	WB	Command cache to writeback block specified by address
Transfer	TR	Command cache to transfer cache block specified by address to another cache
Set State & Writeback	ST-WB	Modify coherence state and then command a writeback of cache block specified by address
Set State & Transfer	ST-TR	Modify coherence state and then command cache to send cache block specified by address to another cache
Set State & Transfer & Writeback	ST-TR-WB	Modify coherence state, command cache to send cache block specified by address to another cache, and then command a writeback of cache block specified by address

Table 3: BedRock Command Network Messages

standard Shared coherence state. This hint is useful when a cache knows that it will never need write permissions for a block and allows the directory to issue the block with read-only permissions instead of read-write permissions. Instruction caches typically issue non-exclusive read requests because the cache does not have the ability to perform writes and modify instruction memory. BedRock does not call a non-exclusive read request an instruction fetch request because this type of request may be issued by any cache controller and for either data or instruction memory locations.

A cache controller is not allowed to re-issue a coherence request until the existing request has been processed by the directory and resolved at the cache controller. A write request for a specific cache block may be issued after a read request for the same block, but multiple write or read requests for the same block from the same controller are not allowed.

Readers familiar with directory-based cache coherence may notice that BedRock does not contain a separate Upgrade request message. Upgrades exist in some protocols to indicate that write permissions are needed for a cache block that the cache controller currently has cached with read-only permissions. BedRock omits this message type because it introduces a race into the coherence protocol between the cache issuing the upgrade and any other cache issuing an upgrade or write request for the same block. Section 8.8.1 of [NSH+20] explains this type of coherence protocol race in detail.

Message	Abbreviation	Description
Data	DATA	Provide data and coherence state for block specified by address and wake up cache to complete request

Table 4: BedRock Fill Network Messages

3.3.2 Command Network

Table 3 lists the Command Network messages that the coherence directory may send to the cache controllers. Command messages are used to modify the state of cache blocks currently cached at a cache controller, provide new cache blocks to a cache, and evict blocks from a cache when a replacement is required to make room for a newly requested block. Every command generates a response of some form from the cache controller. The STW and DATA trigger the sending of a CohAck response, a WB triggers either a DirtyWB or NullWB response, an Inv triggers an InvAck response, and a TR triggers the sending of a DATA fill message to another cache controller.

The ST-WB, ST-TR, and ST-TR-WB messages are compound messages constructed from the set state (ST), TR, and WB message primitives. Semantically, the compound messages perform the indicated primitives in the order listed. That is, a ST-TR-WB causes the cache controller to perform a set state operation, followed by a Transfer to send a DATA fill to another cache, and lastly a WB to send a writeback response to the directory. These operations must happen "atomically" at the cache controller, in that the no other operation or command should occur between any component of the compound command. Abstractly, the compound messages are similar to atomic read-modify-write operations or transactions in that the actions taken must be all-or-none, although there is no possibility that the operations will not happen in the coherence protocol.

BedRock includes these compound messages to reduce network traffic and enhance coherence system performance. They are sent as a single message across the network, and the main benefit of these compound messages is that it simplifies protocol correctness when using unordered networks. On an unordered network it would be possible for a series of ST, TR, and WB commands to arrive out of order at the cache controller, requiring protocol redesign or transient states to handle these race conditions. This makes the protocol significantly more complex and runs against our design goal of simplifying the coherence system.

The cache controllers must process coherence commands, including compound compounds, as atomic operations. This guarantees that the coherence state of every block at the cache controller is only visible in one of the stable MOESIF states at all times. In practice this means that coherence commands need to be serialized with cache accesses such that a cache access cannot see a block's metadata or data in different states during its lifetime.

3.3.3 Fill Network

Table 4 lists the Fill Network messages that a cache controller may send to another cache controller. Currently, the Fill network carries only DATA messages that send a full cache block and its associated state and tag to the destination controller. The state and tag portions of the message are provided by the coherence directory in an arriving Transfer command. Unlike the other coherence networks, the cache controller must support both sending and receiving messages on the Fill network.

The cache controllers must process coherence fill messages as atomic operations. This guarantees

Message	Abbreviation	Description
Invalidate Ack	InvAck	Response to an Invalidate command
Coherence Ack	CohAck	Response to finish coherence transaction
Writeback	DirtyWB	Response to Writeback command with cache block data
Null Writeback	NullWB	Response to Writeback command with no data

Table 5: BedRock Response Network Messages

that the coherence state of every block at the cache controller is only visible in one of the stable MOESIF states at all times. In practice this means that updates to the cache data and metadata from fill messages must be serialized with cache accesses such that a cache access cannot see a block's metadata or data in different states during its lifetime.

3.3.4 Response Network

Table 5 lists the Response Network messages that the cache controller may send to the coherence directory. Responses triggered by the receipt of a Command message. As noted above, STW and DATA commands result in a CohAck being sent, Inv commands trigger an InvAck, and WB commands trigger either DirtyWB or NullWB. Only DirtyWB messages carry data on the Response Network. As described in Section 2.2, the Response network is the highest priority network. In terms of the protocol this means that messages on the response network should receive priority over command or request messages for processing by the coherence directory, and response messages do not cause any other messages to be sent.

3.4 Coherence Transactions and Tracking Coherence State

The BedRock coherence protocol relies on the concept of a Coherence Transaction to define the concurrency behavior of the protocol. BedRock also relies on the concepts of Tag Sets and Way Groups to track coherence state and enforce ordering among related coherence transactions. This section first defines what a coherence transaction is, in both general and controller-specific terms, and then explains how the controllers track and enforce coherence state and coherence transaction order.

3.4.1 Coherence Transaction

In general, a coherence transaction encompasses the total duration of a coherence request, beginning with the cache controller issuing a Request message to the directory and ending when the coherence directory receives the Coherence Acknowledgment (CohAck) response from the cache controller.

3.4.2 Cache Controller

At the cache controller, a coherence transaction begins when it issues a coherence request to the coherence directory. The coherence transaction completes when the controller receives either a STW command or a DATA message (on Command or Fill) and issues a CohAck response to the coherence directory.

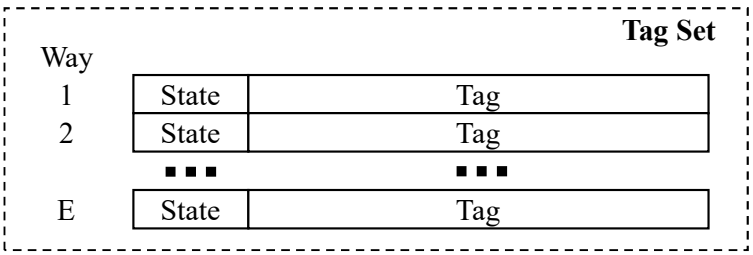

Figure 5: Tag Set

3.4.3 Coherence Directory

At the coherence directory, a coherence transaction begins when it starts processing a new coherence request message. The coherence transaction completes when the directory receives a CohAck response message from the cache controller that initiated the coherence request.

3.4.4 Tracking State - Tag Sets

The coherence state of every block cached in the system is tracked using the concept of a Tag Set, which is depicted in Figure 5. A Tag Set is simply the collection of address tag and coherence state for each cache block (way) within a single cache set. The pair of address tag and coherence state is called a Tag Set Entry. The cache controller has one Tag Set per cache set present in the cache it manages, and typical implementations integrate the tracking of coherence state into the existing address tag metadata associated with each cache block. The coherence directory tracks all of the Tag Sets for every cache set at each cache controller in the coherence system. The directory's collection of Tag Sets comprises the full-duplicate tag directory.

At all times, the Tag Sets tracked at the coherence directory are considered to be the *golden* copies of the Tag Sets, which hold the current state of the coherence system across all cached blocks. The directory updates its Tag Sets during request processing, primarily when sending commands that change the coherence state of blocks at the cache controllers[1]. In contrast to the directory, the cache controllers maintain *shadow* copies of the Tag Sets that are read-only by the cache controller[2]. The cache controller tag sets are updated by coherence commands received by the controller.

3.4.5 Ordering Transactions - Way Groups

The coherence directory is organized as a full-duplicate tag directory that precisely tracks and manages all of the Tag Sets from every cache controller. All of the Tag Sets in the system are grouped into Way Groups that provide coherence transaction ordering for related addresses. Each Way Group allows for a single active coherence transaction at a time, but coherence transactions in different Way Groups are independent and may be processed concurrently.

Figure 6 depicts the contents of a Way Group, which contains one Tag Set from every cache controller in the coherence system and a Pending Bit. In the canonical BedRock system where every cache controller has the same organization and there are S sets per cache, there are S Way Groups at the coherence directory with Way Group X including Tag Set X from every cache controller. In other words, a cache block that maps to cache set X (equivalently, Tag Set X), is a member of Way

[1]Since the BedRock network guarantees message delivery, the state updates at the directory occur when the directory sends coherence commands

[2]The sole exception to this read-only property is a silent upgrade from E to M to record that a write occurred and a cache block has become dirty.

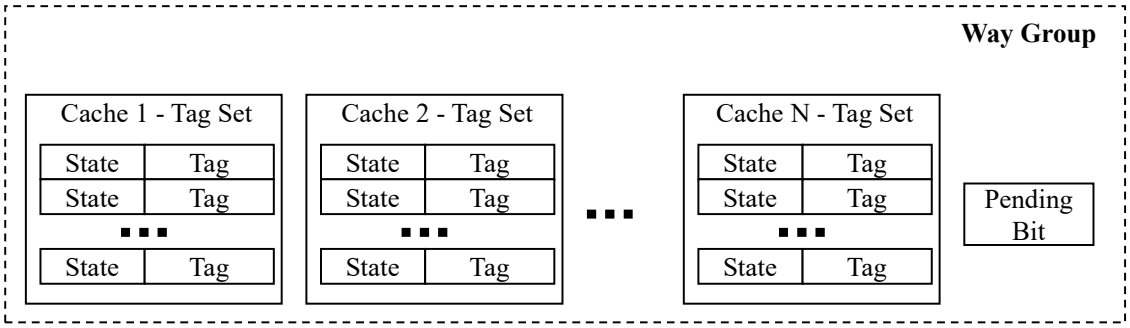

Figure 6: Way Group

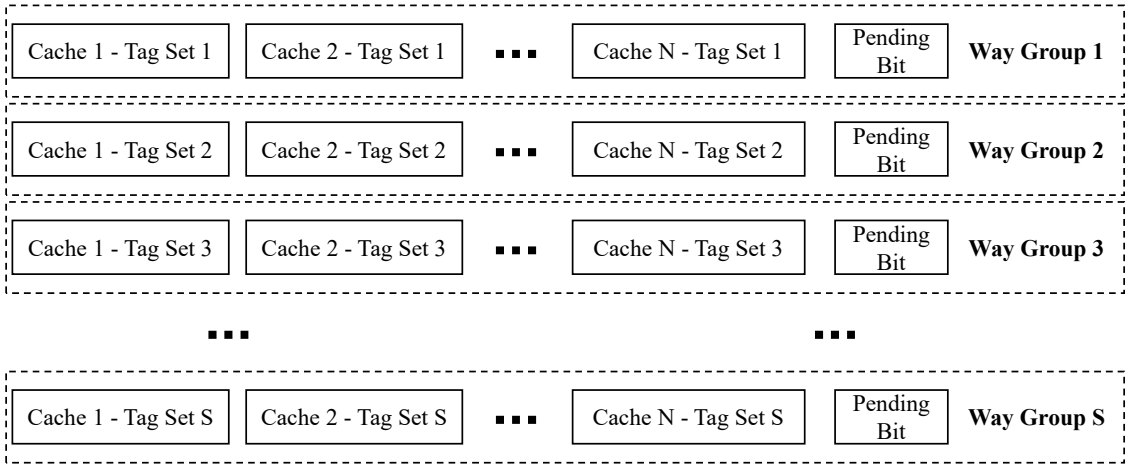

Figure 7: Canonical BedRock Way Groups

Group X. Figure 7 shows the Way Groups of a canonical system with N caches, S sets per cache, and S Way Groups.

The Pending Bit in each Way Group is used to enforce transaction ordering for requests that target the same Way Group. This bit is set when the coherence directory begins processing a coherence request targeting a cache block belonging to the associated Way Group and is cleared when the coherence acknowledgement (CohAck) message for the transaction is received by the directory. Any newly arriving request at the coherence directory must check the pending bit of the target Way Group and stall if the bit is set. Coherence transactions targeting separate Way Groups are, by definition, independent, and may be processed concurrently because all cache blocks that map to the same cache set are a member of the same Way Group. Thus requests to a single Way Group have no possibility of causing any coherence state change to a block in any other Way Group.

3.5 Protocol Assumptions

The BedRock protocol has the following assumptions that must be enforced to ensure correct operation of the coherence protocol.

1. Each Way Group supports one active coherence transaction at a time at the coherence directory.

2. The coherence directory controls all coherence state transitions, with the single exception that a cache controller may silently upgrade a block from Exclusive (E) to Modified (M) on a write operation.

3. The cache controllers must not issue a duplicate coherence request until the original outstanding request is resolved.

4. The cache controllers must process all coherence commands atomically.

3.6 Coherence Protocol Tables

This section presents BedRock's MOESIF cache coherence protocol in tabular form for both the cache controller and coherence directory. The tables use a notation of "Action/State" to describe the behavior of the controllers. Given the current coherence state for a cache block, as indicated at the start of the row, an entry in the table describes the action taken by the controller and the next coherence state of the block at the controller for the event indicated by the column header. If no action is required, a — is written in place of the action. Blank entries indicate that the event listed in the column header cannot occur for the state given at the start of the row. Additional tables for the other protocol variants in the MOESIF family can be found in Appendix A.

3.6.1 Cache Controller Protocol Table

Table 6 presents the cache controller protocol table for the BedRock MOESIF protocol. A cache controller may experience Cache Action and Coherence Message events. Cache Actions are cacheable load and store operations. Coherence Messages are the arrival of a BedRock Command message.

Each "Action/State" entry indicates a message sent in response to the command and the next state of the target block at the cache controller. The Action may be a response message sent to the directory or a command messages sent to another cache controller (as directed by the arriving command from the directory). Some entries in the table have a next coherence state of X that indicates the next coherence state is not known *a priori* by the cache controller. In these situations, the arriving command provides the correct coherence state, which is applied to the block by the cache controller. The specific next state is determined by the coherence directory.

3.6.2 Coherence Directory Protocol Table

Table 7 presents the coherence directory protocol table for the BedRock MOESIF protocol. The coherence directory experiences two types of coherence events: new Coherence Requests from the cache controllers and cache block replacements (a Directory Action) that are initiated by the coherence directory while processing a coherence request. Replacements occur when a cache block in the target cache set of a coherence request must be evicted to make room for the newly requested cache block.

Each "Action/State" entry provides the messages sent by the directory to cache controllers to complete processing of the request and the next state of the target block at the coherence directory. The coherence state superscripts attached to some messages in the table indicate a coherence state associated with the message or compound message component. For example, a ST^{F}-TR^{S}-WB message instructs a cache controller to set the target cache block's state to F, forward that same block to another controller with the S state, and lastly send a writeback to the coherence directory.

Replacements are only required for blocks that may be dirty and are cached in the E, M or O states. Cache blocks in the S or F state are known to be clean and therefore do not require a

separate invalidation or writeback message prior to the new cache block arriving and overwriting the existing block. The block being replaced may be freely used by the cache until it is overwritten or the invalidation and writeback occurs. Dirty blocks must be invalidated to prevent a write racing with the completion of the coherence request.[3] The assumption of one coherence transaction per Way Group at a time ensures that there is no race between multiple coherence requests that could modify the state of the block being replaced.

[3]Dirty blocks in E, M or O technically only require writeback and downgrade to a read-only state. However, we choose to invalidate these blocks to align with the conceptually similar PUT procedure of traditional directory protocols with cache-initiated replacements.

	Cache Action		Coherence Message							
State	Load	Store	Inv	DATA	STW	WB	TR	ST-WB	ST-TR	ST-TR-WB
I	ReqRd	ReqWr		CohAck/X						
S	Hit	ReqWr	InvAck/I		CohAck/M					
E	Hit	Hit/M				NullWB/E		NullWB/X	DATA/X	DATA, NullWB/X
M	Hit	Hit				DirtyWB/M		DirtyWB/X	DATA/X	DATA, DirtyWB/X
O	Hit	ReqWr			CohAck/M	DirtyWB/O	DATA/O	DirtyWB/X	DATA/X	
F	Hit	ReqWr			CohAck/M		DATA/F		DATA/X	

Table 6: BedRock Cache Controller Protocol Table - MOESIF

	Coherence Request					Directory Action
Directory State	ReqRd	ReqRd (Non-Excl)	ReqWr from Invalid	ReqWr from Sharer	ReqWr from Owner	Replacement
I	DATA to Req/E	DATA to Req/S	DATA to Req/M			
S	DATA to Req/S	DATA to Req/S	Inv all S, DATA to Req/M	Inv all S, STW^{M} to Req/M		
E	ST^{F}-TR^{S}-WB to Owner/F	ST^{F}-TR^{S}-WB to Owner/F	ST^{I}-TR^{M} to Owner/M			ST^{I}-WB to Req/I
M	ST^{O}-TR^{S} to Owner/O	ST^{O}-TR^{S} to Owner/O	ST^{I}-TR^{M} to Owner/M			ST^{I}-WB to Req/I
O	TR^{S} to Owner/O	TR^{S} to Owner/O	Inv all S, ST^{I}-TR^{M} to Owner/M	Inv other S and Owner, STW^{M} to Req/M	Inv all S, STW^{M} to Req/M	ST^{I}-WB to Req/I
F	TR^{S} to Owner/F	TR^{S} to Owner/F	Inv all S, ST^{I}-TR^{M} to Owner/M	Inv other S and Owner, STW^{M}/M	Inv all S, STW^{M} to Req/M	

Table 7: BedRock Coherence Directory Protocol Table - MOESIF

Event	Current State	Next State
Load	I	S, E
Store	I, S, O, F	M
Store (Silent Upgrade)	E	M
Other Load	E	F
	M	O
Other Store	S, E, M, O, F	I

Table 8: BedRock Cache Controller Next State Table - MOESIF

3.7 Coherence State Transitions

This section describes the possible coherence state transitions at the cache and directory controllers for the BedRock MOESIF protocol. Each table describes the coherence state transitions for a single cache block given a current starting state and a cache or coherence event that causes a state transition. Events that do not cause a change in the coherence state at the controller are not listed.

3.7.1 Cache Controller State Transitions

Table 8 enumerates the possible coherence state transitions as observed by the cache controller for a single cache block as loads and stores occur in the coherence system targeting that block. The Current State column lists the current cache coherence state of the target block and the Next State column provides the next state of the block at the cache controller. The Event column lists the type of load or store event. Events that do not cause a change in the coherence state are not listed (e.g., Load to block in S remains in S).

The first three rows correspond to actions taken by the controller itself, while the last two rows of Other Load and Other Store correspond to load and store actions initiated by some other cache controller. A Store (Silent Upgrade) occurs when the cache performs a store operation on a block cached in the E state. This state has read and write permissions but is considered clean, therefore the store must transition the block to the M state to indicate a write has occurred. This is called a Silent Upgrade since the cache controller does not need to notify the coherence directory of the write because it already has write permissions for the block.

3.7.2 Coherence Directory State Transitions

Table 9 describes the possible coherence state transitions at the coherence directory for a single cache block as load and store misses are processed. For each event type we list the corresponding BedRock Request Network message type received by the coherence directory, followed by the current state of the coherence directory and the resulting next state of the block at both the directory and the cache controller that initiated the event. This table fully enumerates the possible state transitions for the MOESIF protocol, covering the cross-product of events and current directory states. In accordance with the SWMR Invariant, a store operation always results in a single cache owning the block and receiving write permissions.

As seen in the table, even when using a MOESIF protocol, the requesting cache will only ever receive a cache block in the S, E, or M states. The O and F states are used when load requests

Event	BedRock Request Message	Current State (Dir)	Next State (Dir)	Next State (Requestor)
Load	ReqRd	I	E	E
		S	S	S
		E, F	F	S
		M, O	O	S
Load (Non-Excl)	ReqRd (Non-Excl)	I, S	S	S
		E, F	F	S
		M, O	O	S
Store	ReqWr	I, S, O, E, M, F	M	M

Table 9: BedRock Coherence Directory Next State Table - MOESIF

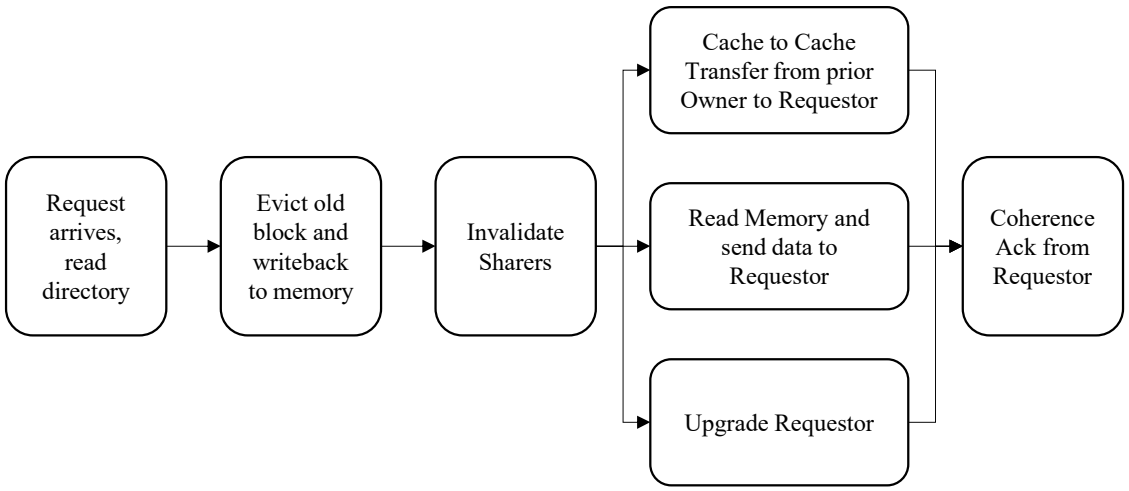

Figure 8: Generic Coherence Directory Request Processing Flow

target cache blocks in either the M or E states, respectively. These states allow a single cache controller to retain ownership permissions for read-only cache blocks and allow a read miss to be completed with a cache to cache transfer rather than a LLC or main memory access.

Note that an event may cause the owner cache to change during the transaction, even if the block's state at the directory does not change For example, one cache performing a write to a block that another cache already has write permissions for (i.e., cached in M) results in the state remaining in M but changes ownership of the block.

3.8 Coherence Directory Processing Flow

Figure 3 depicts a generic, high-level BedRock coherence directory request processing flow. In general, every request first checks the pending bit of the request's associated Way Group and then reads the directory if there is no active coherence transaction for that Way Group. If the directory detects that a dirty cache block needs to be evicted and replaced from the requesting cache controller to make room for the requested block a replacement and writeback is initiated.

Next, the directory invalidates all sharers of the block if the request is a write to maintain the SWMR Invariant. A controller in the O or F state may also be invalidated as indicated in Table 7 for certain write requests. Next, the directory either fetches the required block from memory or an LLC, initiates a cache to cache transfer of the block (possibly modifying the state of the current owner and writing back the block if needed), or sends a STW message to the requestor to upgrade its permissions if it already has an up-to-date copy of the block. Lastly, the directory waits for the coherence acknowledgment response from the requesting cache controller to finalize the transaction and clear the pending bit. The coherence directory updates its Tag Sets as needed when issuing commands that change the coherence state of cache blocks in the system.

The presented processing flow encompasses the necessary logical steps to process a new coherence request. It is up to the specific coherence directory implementations to optimize this flow for concurrency and performance.

3.9 Coherence Protocol Comparison

BedRock is similar to, but subtly different from, a traditional directory-based coherence protocol. In this section, we compare the BedRock protocol to the traditional directory protocols presented in Sections 8.3 and 8.4 of [NSH+20].

3.9.1 Transient States

Transient states exist in traditional directory-based coherence protocols to handle race conditions within the protocol. BedRock does not expose any transient states in the coherence protocol. BedRock is able to realize a protocol without exposed transient states due to its assumptions that only a single coherence transaction per way group may be active at any time and that each cache controller must not issue duplicate coherence requests. The cache controllers are able to use cache blocks with their existing permissions, but must request new permissions from the directory as needed. Coherence commands are processed atomically by the cache controllers to guarantee that cache accesses only ever see a cache block in a single stable and consistent state. The coherence directory updates the stable state of a cache block as it issues messages to change the state of the block. Since the directory only processes one request per way group at a time there is no need to expose transient states in the protocol. Implementations may define mechanisms to track the transient behavior of the request processing flow (e.g., invalidations completed, waiting for coherence acknowledgment) in order to enable concurrent processing of independent requests, but at the protocol level all blocks will be in a consistent and stable state at all times.

3.9.2 Protocol Messages

Table 10 presents a comparison of the message types in BedRock and those used in the traditional MESI protocol. The corresponding message types of the traditional protocol are found in Tables 6.4, 8.3, 8.4, 8.5, and 8.6 of [NSH+20]. A Fwd-GetX message corresponds to a Fwd-Get message with state specified by X, where X is one of the MOESIF states as applicable for a specific instance of the message (e.g., Fwd-GetS or Fwd-GetM). The PUT messages have also been extended to cover all of the MOESI states.

As seen in the table, many BedRock messages have direct equivalences in the traditional protocol. However, a few important differences are worth discussing. First, BedRock carries coherence messages on four networks instead of three, so there is not a one-to-one correspondence to the three networks of the traditional protocol, which are called Request, Forwarded-Request, and Response.

BedRock Network	BedRock Message	Message in [NSH+20]
Request	ReqRd	GetE
	ReqRd (Non-Excl)	GetS
	ReqWr	GetM or Upgrade
Command	Inv	Inv
	DATA	Data from Dir or Owner
	STW	Data from Dir (ack=0)
	WB	No direct equivalence
	TR	Fwd-GetX
	ST-WB	No direct equivalence
	ST-TR	Fwd-GetX
	ST-TR-WB	Fwd-GetX
Response	InvAck	Inv-Ack
	CohAck	No direct equivalence
	DirtyWB	PUTM, PUTO
	NullWB	PUTS, PUTE, PUTF

Table 10: BedRock Protocol Message Equivalency

The Request network is similar in each protocol as they carry coherence requests from cache controller to coherence directory. BedRock's Command network is similar to the Forwarded-Request network as both networks carry messages from the coherence directory to the cache controller. Bedrock's Response and Fill networks carry messages similar to the traditional protocol's Response network.

Second, there is no cache controller Replacement message that can be issued on BedRock's Request network. Unlike a traditional protocol, the coherence directory, not the cache controller, is responsible for performing cache block replacements to make room for new cache blocks in the cache controller. The cache controller provides a replacement way "hint" to the coherence directory with each read or write request. The coherence directory then finalizes the selection of a replacement way, writing back any dirty cache block data as required by issuing a WB message over the Command Network. BedRock's directory-controlled replacements eliminate a common race between PUT and Fwd-Get messages in the traditional protocol. Thus, one view of the WB message is that it is a directory-initiated PUT and that PUT messages are from directory to cache rather than cache to directory as in the traditional protocol.

Third, the ST-WB message has no direct equivalence in the traditional protocol. ST-WB can be viewed as a directory-initiated PUT message for a dirty cache block. The intent of these messages is to downgrade the permissions of a particular cache block, writing back the dirty block, and then modifying the coherence state of the block. ST-WB is commonly used to combine an invalidate and writeback sequence into a single command/ack pair since the DirtyWB or NullWB response serves as an acknowledgement of both the writeback and invalidation actions.

Lastly, the traditional protocol does not require a CohAck message to close a coherence trans-

action. This message is required to enforce correct serialization of coherence requests targeting the same cache block at the directory controller. The use of an explicit coherence transaction acknowledgement disallows concurrent transactions to the same block and enables the removal of many transient states from the coherence protocol required to handle races at the cache controller.

3.9.3 Coherence Networks

BedRock utilizes four coherence networks instead of the three networks used by the comparison protocol. In Bedrock, only the Fill network is bi-directional. The Request, Command, and Response networks are all uni-directional and carry messages between the cache controllers and the coherence directory. BedRock assumes that there may only be one active coherence transaction per way group, which necessitates the use of a coherence acknowledgement message from cache to directory to close the transaction. This requires a fourth coherence message class that is higher priority than the traditional protocol's existing Response network. A transaction involving a cache to cache transfer requires four phases or hops: Request to directory, Command to owner, Fill from owner to requestor, Response to directory. The primary benefit of BedRock's single transaction per way group assumption is that it eliminates many possible transient states found in the traditional protocol. The use of primarily uni-directional networks and the single transaction assumption have proven beneficial in simplifying the on-chip networks of BedRock's initial implementation in BlackParrot.

Protocol	Cache Count					
	2	3	4	5	6	8
BedRock	0.1s	0.21s	3.1s	47s	8.9m	15.2h
Traditional	0.1s	0.35s	19.9s	10.4m	9.9h	175d
Speedup	1.0	1.6	6.4	13.3	66.6	1230

Table 11: CMurphi verification time and speedup - MESI protocols

4 BedRock Cache Coherence Protocol Verification

The BedRock cache coherence protocol is similar to, yet subtly different from, commonly understood directory-based MESI protocols (e.g., Section 8.3 of [NSH+20]). Therefore, it is important that the protocol itself is shown to be correct, especially since it is implemented by the BlackParrot multicore. BedRock is verified correct using CMurphi [PIM+04], an improved version of the Murphi [DDH+04] model checking framework. BedRock's CMurphi description assumes that only a single cache block and a single coherence directory are modeled. These assumptions are valid because, by definition, cache coherence is constrained to a single memory location, BedRock's coherence directories operate independently from one another in a multi-directory system, and every cache block is managed by a single directory.

Table 11 shows the verification time required by CMurphi for BedRock and a traditional MESI coherence protocol, averaged over three runs for each configuration. Verification time for the traditional MESI protocol at 8-caches is a best-fit estimate, as it would take 175 days to complete! Both protocols are verified correct without error by CMurphi for all other configurations. CMurphi explores considerably fewer states and completes verification significantly faster for BedRock, up to 66x faster for a 6-cache system. The verification speedup is a direct consequence of BedRock's design decisions eliminating protocol races and transient states, which greatly reduces the necessary state-space that must be explored and verified.

5 Uncached and Atomic Accesses in BedRock

This section describes how uncached and atomic read-modify-write accesses are handled in BedRock. We continue to assume the canonical address space layout and cacheability properties presented in Section 3.1 and Figure 4.

5.1 Uncacheable Accesses

Uncacheable accesses may target both uncacheable and cacheable memory regions. Uncacheable accesses to uncacheable memory are not a concern for the BedRock coherence protocol because the protocol enforces coherence only for cacheable memory. Uncacheable accesses to uncacheable memory can either bypass the coherence directory or the directory can be augmented to forward the requests and responses to and from memory, respectively. Thus, the only changes required to support uncacheable accesses to uncacheable memory are to add appropriate request and command message types for the BedRock networks and modify the system to handle these requests, assuming that the accesses will travel on the existing BedRock networks. Endpoints that do not participate in coherence must only implement the Request and Command BedRock networks.

Uncacheable accesses to cacheable memory must participate in the BedRock coherence protocol to guarantee coherence within the system. An uncacheable access targeting a cacheable block of memory first must invalidate, and write back, the block from all cache controllers that have it cached. Then, the uncacheable access may me issued to memory with the response from memory being forwarded back to the requesting cache. This access must also be serialized with all coherence requests to the target cache block. Serializing the request is easily enforced using the existing way group and pending bit mechanisms of the coherence directory that serialize cacheable accesses. BedRock must be modified to support the uncacheable request message types and corresponding command message types that deliver uncacheable load data or store complete confirmation to the requesting cache controller. The coherence directory must also be modified to detect an uncacheable request targeting cacheable memory and invoke the invalidation and writeback routines for the target cache block.

5.2 Atomic Read-Modify-Write Operations

Atomic read-modify-write style operations, atomics for short, are important operations for multicore processors. Consequently, a multicore processor's memory system, including the coherence system, must support these operations. In the context of the canonical BedRock coherence system, there are two possible locations that atomics can be executed — at the LLC/memory or within the cache controller managed inclusive cache hierarchy. We assume that an atomic to uncacheable memory is executed at the LLC/memory while an atomic to cacheable memory is executed by either the LLC/memory or the cache controller's cache hierarchy. This is the target model used by BedRock in BlackParrot [PGW+20].

BedRock easily supports atomic operations targeting cacheable memory and executed by the cache controller's inclusive cache hierarchy with very minimal modification to the existing cache controller. A cache executing an atomic simply needs write permissions for the target cache block. Once it has write permissions it must complete the read-modify-write sequence as a single, uninterruptible action. The simplest way to accomplish this is for the cache to perform a write request for the target block then briefly "lock" the block while the cache completes the read-modify-write operation. After the atomic executes, the block is unlocked and any coherence commands that arrived targeting the block are processed. Although this seems to violate the spirit of BedRock in

that the cache controller momentarily ignores a command from the directory, it is an effective way to implement atomics in practice. As long as the block is locked for only a very short time, there is no risk of deadlocking the coherence protocol. From the point of view of the coherence directory, a controller locking a block for a few cycles is equivalent to the coherence network taking a few extra cycles to deliver a coherence command. Since BedRock does not depend on the coherence networks delivering messages within specific latencies, the few cycles of delay from locking has no impact on correctness, so long as this delay is short and the cache controller resumes processing commands in a timely manner.

Executing atomics at LLC/memory is similar to performing an uncached memory access. The cache controller issues an atomic request with data that must be forwarded to memory for use in the read-modify-write operation, and memory responds with data if the atomic has a return value or an atomic complete message if there is not return value. The memory response is then forwarded to the requesting cache controller on the BedRock command network. This type of operation requires adding a few new message types to the BedRock request and command networks to support atomic requests and atomic data or complete commands. Atomics targeting cacheable memory and executed by the LLC/memory must follow a procedure similar to a regular uncached access to cacheable memory that invalidates and writes back the target cache block from all cache controllers possessing a copy of it. This forces any cache currently using the block to refetch it from memory, and these requests will be serialized by the coherence directory to guarantee the Data Value Invariant holds. Thus, the coherence directory must be modified in a similar manner as it was for uncacheable requests to detect atomic accesses to cacheable memory and enforce serialization of these requests using the existing way group and pending bit mechanisms.

A BedRock Cache Coherence Protocol Tables

A.1 Cache Controller Protocol Tables

	Cache Action		Coherence Message							
State	Load	Store	Inv	DATA	STW	WB	TR	ST-WB	ST-TR	ST-TR-WB
I	ReqRd	ReqWr		CohAck/X						
M	Hit	Hit				DirtyWB/M		DirtyWB/X	DATA/X	DATA, DirtyWB/X

Table 12: BedRock Cache Controller Protocol Table - MI

	Cache Action		Coherence Message							
State	Load	Store	Inv	DATA	STW	WB	TR	ST-WB	ST-TR	ST-TR-WB
I	ReqRd	ReqWr		CohAck/X						
S	Hit	ReqWr	InvAck/I		CohAck/M					
M	Hit	Hit				DirtyWB/M		DirtyWB/X	DATA/X	DATA, DirtyWB/X

Table 13: BedRock Cache Controller Protocol Table - MSI

	Cache Action		Coherence Message							
State	Load	Store	Inv	DATA	STW	WB	TR	ST-WB	ST-TR	ST-TR-WB
I	ReqRd	ReqWr		CohAck/X						
S	Hit	ReqWr	InvAck/I		CohAck/M					
E	Hit	Hit/M				NullWB/E		NullWB/X	DATA/X	DATA, NullWB/X
M	Hit	Hit				DirtyWB/M		DirtyWB/X	DATA/X	DATA, DirtyWB/X

Table 14: BedRock Cache Controller Protocol Table - MESI

	Cache Action		Coherence Message							
State	Load	Store	Inv	DATA	STW	WB	TR	ST-WB	ST-TR	ST-TR-WB
I	ReqRd	ReqWr		CohAck/X						
S	Hit	ReqWr	InvAck/I		CohAck/M					
E	Hit	Hit/M				NullWB/E		NullWB/X	DATA/X	DATA, NullWB/X
M	Hit	Hit				DirtyWB/M		DirtyWB/X	DATA/X	DATA, DirtyWB/X
F	Hit	ReqWr			CohAck/M		DATA/F		DATA/X	

Table 15: BedRock Cache Controller Protocol Table - MESIF

	Cache Action		Coherence Message							
State	Load	Store	Inv	DATA	STW	WB	TR	ST-WB	ST-TR	ST-TR-WB
I	ReqRd	ReqWr		CohAck/X						
S	Hit	ReqWr	InvAck/I		CohAck/M					
M	Hit	Hit				DirtyWB/M		DirtyWB/X	DATA/X	DATA, DirtyWB/X
O	Hit	ReqWr			CohAck/M	DirtyWB/O	DATA/O	DirtyWB/X	DATA/X	

Table 16: BedRock Cache Controller Protocol Table - MOSI

	Cache Action		Coherence Message							
State	Load	Store	Inv	DATA	STW	WB	TR	ST-WB	ST-TR	ST-TR-WB
I	ReqRd	ReqWr		CohAck/X						
S	Hit	ReqWr	InvAck/I		CohAck/M					
M	Hit	Hit				DirtyWB/M		DirtyWB/X	DATA/X	DATA, DirtyWB/X
O	Hit	ReqWr			CohAck/M	DirtyWB/O	DATA/O	DirtyWB/X	DATA/X	
F	Hit	ReqWr			CohAck/M		DATA/F		DATA/X	

Table 17: BedRock Cache Controller Protocol Table - MOSIF

	Cache Action		Coherence Message							
State	Load	Store	Inv	DATA	STW	WB	TR	ST-WB	ST-TR	ST-TR-WB
I	ReqRd	ReqWr		CohAck/X						
S	Hit	ReqWr	InvAck/I		CohAck/M					
E	Hit	Hit/M				NullWB/E		NullWB/X	DATA/X	DATA, NullWB/X
M	Hit	Hit				DirtyWB/M		DirtyWB/X	DATA/X	DATA, DirtyWB/X
O	Hit	ReqWr			CohAck/M	DirtyWB/O	DATA/O	DirtyWB/X	DATA/X	

Table 18: BedRock Cache Controller Protocol Table - MOESI

A.2 Coherence Directory Protocol Tables

	Coherence Request					Directory Action
Directory State	ReqRd	ReqRd (Non-Excl)	ReqWr from Invalid	ReqWr from Sharer	ReqWr from Owner	Replacement
I	DATA to Req/M	DATA to Req/M	DATA to Req/M			
M	ST^{I}-TR^{M} to Owner/M	ST^{I}-TR^{M} to Owner/M	ST^{I}-TR^{M} to Owner/M			ST^{I}-WB to Req/I

Table 19: BedRock Coherence Directory Protocol Table - MI

	Coherence Request					Directory Action
Directory State	ReqRd	ReqRd (Non-Excl)	ReqWr from Invalid	ReqWr from Sharer	ReqWr from Owner	Replacement
I	DATA to Req/S	DATA to Req/S	DATA to Req/M			
S	DATA to Req/S	DATA to Req/S	Inv all S, DATA to Req/M	Inv all S, STW^{M} to Req/M		
M	ST^{S}-TR^{S}-WB to Owner/S	ST^{S}-TR^{S}-WB to Owner/S	ST^{I}-TR^{M} to Owner/M			ST^{I}-WB to Req/I

Table 20: BedRock Coherence Directory Protocol Table - MSI

	Coherence Request					Directory Action
Directory State	ReqRd	ReqRd (Non-Excl)	ReqWr from Invalid	ReqWr from Sharer	ReqWr from Owner	Replacement
I	DATA to Req/E	DATA to Req/S	DATA to Req/M			
S	DATA to Req/S	DATA to Req/S	Inv all S, DATA to Req/M	Inv all S, STW^{M} to Req/M		
E	ST^{S}-TR^{S}-WB to Owner/S	ST^{S}-TR^{S}-WB to Owner/S	ST^{I}-TR^{M} to Owner/M			ST^{I}-WB to Req/I
M	ST^{S}-TR^{S}-WB to Owner/S	ST^{S}-TR^{S}-WB to Owner/S	ST^{I}-TR^{M} to Owner/M			ST^{I}-WB to Req/I

Table 21: BedRock Coherence Directory Protocol Table - MESI

	Coherence Request					Directory Action
Directory State	ReqRd	ReqRd (Non-Excl)	ReqWr from Invalid	ReqWr from Sharer	ReqWr from Owner	Replacement
I	DATA to Req/E	DATA to Req/S	DATA to Req/M			
S	DATA to Req/S	DATA to Req/S	Inv all S, DATA to Req/M	Inv all S, STW^{M} to Req/M		
E	ST^{F}-TR^{S}-WB to Owner/F	ST^{F}-TR^{S}-WB to Owner/F	ST^{I}-TR^{M} to Owner/M			ST^{I}-WB to Req/I
M	ST^{F}-TR^{S}-WB to Owner/F	ST^{F}-TR^{S}-WB to Owner/F	ST^{I}-TR^{M} to Owner/M			ST^{I}-WB to Req/I
F	TR^{S} to Owner/F	TR^{S} to Owner/F	Inv all S, ST^{I}-TR^{M} to Owner/M	Inv other S and Owner, STW^{M}/M	Inv all S, STW^{M} to Req/M	

Table 22: BedRock Coherence Directory Protocol Table - MESIF

	Coherence Request					Directory Action
Directory State	ReqRd	ReqRd (Non-Excl)	ReqWr from Invalid	ReqWr from Sharer	ReqWr from Owner	Replacement
I	DATA to Req/S	DATA to Req/S	DATA to Req/M			
S	DATA to Req/S	DATA to Req/S	Inv all S, DATA to Req/M	Inv all S, STW^{M} to Req/M		
M	ST^{O}-TR^{S} to Owner/O	ST^{O}-TR^{S} to Owner/O	ST^{I}-TR^{M} to Owner/M			ST^{I}-WB to Req/I
O	TR^{S} to Owner/O	TR^{S} to Owner/O	Inv all S, ST^{I}-TR^{M} to Owner/M	Inv other S and Owner, STW^{M} to Req/M	Inv all S, STW^{M} to Req/M	ST^{I}-WB to Req/I

Table 23: BedRock Coherence Directory Protocol Table - MOSI

	Coherence Request					Directory Action
Directory State	ReqRd	ReqRd (Non-Excl)	ReqWr from Invalid	ReqWr from Sharer	ReqWr from Owner	Replacement
I	DATA to Req/F	DATA to Req/S	DATA to Req/M			
S	DATA to Req/S	DATA to Req/S	Inv all S, DATA to Req/M	Inv all S, STW^{M} to Req/M		
M	ST^{O}-TR^{S} to Owner/O	ST^{O}-TR^{S} to Owner/O	ST^{I}-TR^{M} to Owner/M			ST^{I}-WB to Req/I
O	TR^{S} to Owner/O	TR^{S} to Owner/O	Inv all S, ST^{I}-TR^{M} to Owner/M	Inv other S and Owner, STW^{M} to Req/M	Inv all S, STW^{M} to Req/M	ST^{I}-WB to Req/I
F	TR^{S} to Owner/F	TR^{S} to Owner/F	Inv all S, ST^{I}-TR^{M} to Owner/M	Inv other S and Owner, STW^{M}/M	Inv all S, STW^{M} to Req/M	

Table 24: BedRock Coherence Directory Protocol Table - MOSIF

Note: it may be beneficial to make F the next state of a ReqRd to a block in S.

	Coherence Request					Directory Action
Directory State	ReqRd	ReqRd (Non-Excl)	ReqWr from Invalid	ReqWr from Sharer	ReqWr from Owner	Replacement
I	DATA to Req/E	DATA to Req/S	DATA to Req/M			
S	DATA to Req/S	DATA to Req/S	Inv all S, DATA to Req/M	Inv all S, STW^{M} to Req/M		
E	ST^{O}-TR^{S} to Owner/O	ST^{O}-TR^{S} to Owner/O	ST^{I}-TR^{M} to Owner/M			ST^{I}-WB to Req/I
M	ST^{O}-TR^{S} to Owner/O	ST^{O}-TR^{S} to Owner/O	ST^{I}-TR^{M} to Owner/M			ST^{I}-WB to Req/I
O	TR^{S} to Owner/O	TR^{S} to Owner/O	Inv all S, ST^{I}-TR^{M} to Owner/M	Inv other S and Owner, STW^{M} to Req/M	Inv all S, STW^{M} to Req/M	ST^{I}-WB to Req/I

Table 25: BedRock Coherence Directory Protocol Table - MOESI

B BedRock Cache Coherence State Transition Tables

B.1 Cache Controller State Transition Tables

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

Table 26: BedRock Cache Controller Next State Table - MI

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

Table 27: BedRock Cache Controller Next State Table - MSI

Event	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

Table 28: BedRock Cache Controller Next State Table - MESI

Event	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

Table 29: BedRock Cache Controller Next State Table - MESIF

Event	Current State	Next State
Load	I	S
Store	I, S, O	M
Other Load	M	O
Other Store	S, O, M	I

Table 30: BedRock Cache Controller Next State Table - MOSI

Event	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

Table 31: BedRock Cache Controller Next State Table - MOSIF

Event	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

Table 32: BedRock Cache Controller Next State Table - MOESI

B.2 Coherence Directory State Transition Tables

Event	Current State (Dir)	Next State (Dir)	Next State (Requestor)
Load (Any)	I, M	M	M
Store	I, M	M	M

Table 33: BedRock Coherence Directory Next State Table - MI

Event	Current State (Dir)	Next State (Dir)	Next State (Requestor)
Load (Any)	I, S, M	S	S
Store	I, S, M	M	M

Table 34: BedRock Coherence Directory Next State Table - MSI

Event	Current State (Dir)	Next State (Dir)	Next State (Requestor)
Load	I	E	E
	S, E, M	S	S
Load (Non-Excl)	I, S, E, M	S	S
Store	I, S, E, M	M	M

Table 35: BedRock Coherence Directory Next State Table - MESI

Event	Current State (Dir)	Next State (Dir)	Next State (Requestor)
Load	I	E	E
	S, M	S	S
	E, F	F	S
Load (Non-Excl)	I, S, M	S	S
	E, F	F	S
Store	I, S, E, M, F	M	M

Table 36: BedRock Coherence Directory Next State Table - MESIF

Event	Current State (Dir)	Next State (Dir)	Next State (Requestor)
Load (Any)	I, S	S	S
	M, O	O	S
Store	I, S, O, M	M	M

Table 37: BedRock Coherence Directory Next State Table - MOSI

Event	Current State (Dir)	Next State (Dir)	Next State (Requestor)
Load (Any)	I	F	F
	S	S	S
	M, O	O	S
	F	F	S
Store	I, S, O, M, F	M	M

Table 38: BedRock Coherence Directory Next State Table - MOSIF

Event	Current State (Dir)	Next State (Dir)	Next State (Requestor)
Load	I	E	E
	S, E	S	S
	M, O	O	S
Load (Non-Excl)	I, S, E	S	S
	M, O	O	S
Store	I, S, O, E, M	M	M

Table 39: BedRock Coherence Directory Next State Table - MOESI

References

[DDH+04] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. "Protocol verification as a hardware design aid". In: *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors*. IEEE Comput. Soc. Press, 2004. DOI: 10.1109/iccd.1992.276232. URL: https://doi.org/10.1109/iccd.1992.276232.

[NSH+20] 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.

[PIM+04] Giuseppe Della Penna, Benedetto Intrigila, Igor Melatti, Enrico Tronci, and Marisa Venturini Zilli. "Exploiting transition locality in automatic verification of finite-state concurrent systems". In: *International Journal on Software Tools for Technology Transfer* 6.4 (July 2004), pp. 320–341. DOI: 10.1007/s10009-004-0149-6. URL: https://doi.org/10.1007/s10009-004-0149-6.

[PGW+20] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, Tavio Guarino, Ajay Joshi, Mark Oskin, and Michael Bedford Taylor. "BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs". In: *IEEE Micro* 40.4 (2020), pp. 93–102. DOI: 10.1109/MM.2020.2996145. URL: https://doi.org/10.1109/MM.2020.2996145.