Determining the Minimum Number of Virtual Networks for Different Coherence Protocols

Weihang Li
Duke University
wl252@duke.edu

Andrés Goens
University of Amsterdam
a.goens@uva.nl

Nicolai Oswald
NVIDIA
noswald@nvidia.com

Vijay Nagarajan
University of Utah
vijay@cs.utah.edu

Daniel J. Sorin
Duke University
sorin@ee.duke.edu

***Abstract*—We revisit the question of how many virtual networks (VNs) are required to provably avoid deadlock in a cache coherence protocol. The textbook way of reasoning about VNs says that the number of VNs depends on the longest chain of message dependencies in the protocol. We show that this conventional wisdom is incorrect and results in a number of virtual networks that is neither necessary nor sufficient for the general system model of an arbitrary interconnection network (ICN) topology and multiple directories. We have created a formalism for modeling coherence protocols and their interactions with ICN queueing. Using that formalism, we have developed an algorithm that (a) determines the minimum number of virtual networks required to avoid deadlock and (b) generates the mappings from message types to virtual networks.**

I. Introduction

Multicore processors must ensure that coherence protocol deadlocks are impossible. A coherence protocol deadlock can occur due to cyclic dependencies between coherence messages. For example, a deadlock can occur if caches C1 and C2 are waiting for responses from each other, but neither will respond until the other one responds first. More complicated deadlocks can arise through cyclic dependencies that involve the routers of the interconnection network (ICN).

There are three approaches to avoiding cache coherence protocol deadlocks. The first two have significant drawbacks or obstacles to deployment, which is why the third is the most commonly used approach.

- Use negative acknowledgments (nacks). A protocol can use nacks to avoid deadlocks (e.g., DASH [1] and FLASH [2]), but the use of nacks risks livelock and starvation that are difficult to provably avoid. We do not consider nacks further in this paper.
- Allow protocol deadlocks and recover from them. There are several schemes that can detect and recover from potential deadlocks or periodically drain the ICN to avert potential deadlocks [3]–[6]. We discuss these approaches in Section VIII, and explain their advantages as well as reasons why they are not (yet) widely deployed. The main limitation of the approach is its complexity and the difficulty of proving that it avoids deadlocks.
- Use *virtual networks*. Virtual networks (VNs) are simply dedicated buffers at the ends of physical network links, where each VN has the capability to buffer at least one message.[1]By assigning different messages to different VNs, we can ensure that certain messages cannot get blocked due to being behind other messages in the same buffer. VNs are the most common approach today. Indeed, today's industrial strength specifications such as CHI [7], CXL [8], and Tilelink [9] all prescribe VNs for avoiding coherence deadlocks.

In this paper, we focus on the solution of VNs. We revisit the question of how many VNs are required to provably avoid deadlocks, and develop an algorithm that determines the minimum number of VNs and generates the mappings from message types to VNs. Since VNs consume a significant amount of the ICN power, it is beneficial to reduce their number [10]. Note that the minimum number of VNs and the generated VN mappings solve the problem of protocol deadlock; these results are orthogonal to the need for virtual channels to avoid routing deadlock.

The textbook way of reasoning about VNs has been to group messages into classes and assign each class to its own VN [11]. The number of these VNs is determined from the longest chain of messages in the protocol. For example, typical directory protocols group their messages into three classes: requests, forwarded requests, and responses. Because a request can trigger the sending of a forwarded request, and a forwarded request can trigger the sending of a response, we have a chain of dependencies: request → forwarded request → response. To avoid a cyclic dependency, the standard solution is to separate these message classes and assign each class a VN.

We show that this conventional wisdom is incorrect. The textbook algorithm for determining VNs results in a number of virtual networks that is *neither necessary nor sufficient* for the general system model of an arbitrary ICN topology and an arbitrary number of directories. We will show examples of protocols for which the textbook algorithm results in too few and too many VNs. Having too few VNs can result in deadlock. Having too many is a waste of router hardware and power, and it can lead to routers that are slower than they could be [4].

[1] Here we distinguish between virtual networks and virtual channels: whereas the former refers to buffers used to avoid protocol deadlock, the latter refer to buffers used to avoid routing deadlock. The two concepts compose, in that a system that requires N virtual networks and C virtual channels, requires NC distinct buffers at the end of each physical link.

We have developed a novel, formal approach to modeling cache coherence protocols and the impact of ICN queuing on them. Using this approach, we have derived a sufficient condition for ensuring that a given protocol cannot deadlock. Leveraging this condition, we have developed an algorithm that (a) determines the minimum number of VNs required to provably avoid deadlocks and (b) generates the mappings from message types to VNs.

Utilizing the insight from our modeling methodology and VN determination algorithm, we have divided coherence protocols into three classes. This classification assumes the most general system model: multiple directories (interleaved by address) and an arbitrary ICN.

Class 1 protocols are prone to deadlock regardless of the VNs, i.e., they have *protocol deadlock*. It is easy to identify such protocols using a model checker.

Class 2 protocols, which to the best of our knowledge have not been identified before, are prone to a deadlock even if every static message is assigned to a unique VN. To avoid a deadlock, these protocols need a new way of allocating VNs—e.g., a VN for every unique cache line. This is obviously not practical, and it is important to ensure that class 2 protocols are not used. Surprisingly, as we show later, this class includes several well-known protocols [11].

Class 3 protocols are useful because they require a constant number of VNs. In fact, we show that all of these protocols, in practice, require exactly two VNs. The industrial CHI [7] protocol belongs to this class, despite its specification stating it requires four VNs. Thus, a practical implication of this work is that open protocol specifications such as CHI need fewer virtual networks than they mandate.

We have confirmed the results of our analysis by model checking a variety of Class 2 and Class 3 protocols. To consider all possible ICN behaviors in a tractable fashion, we developed a novel ICN model for use in model checking.

We make the following contributions:

- We show that conventional wisdom for computing the number of virtual networks is incorrect.
- We create a formal methodology for modeling coherence protocols and their interactions with ICN queuing.
- We develop an algorithm that determines the minimum number of VNs and generates the mappings from message types to VNs.
- We classify protocols into three classes, based on their VN requirements, including one previously undiscovered class.
- We show that, under the assumption of an arbitrary ICN and multiple directories, practical coherence protocols require two VNs. In particular, we are the first to formalize the CHI [7] protocol specification, and we show that two VNs are sufficient to avoid deadlocks.
- We formally verify our results with the Murphi model checker [12], using a novel ICN model that manifests all possible ICN behaviors.

II. System Model and Background

We consider a very general system model, so as to ensure that our results are broadly applicable. In this system model, there are some number of caches and one or more directories. If there are multiple directories, they are assigned to different cache blocks; i.e., a request for block B will be sent to the directory that is the home for block B. The caches and directories communicate over an interconnection network with an arbitrary topology.

For other system models that are more restrictive, our results may be conservative. For example, a protocol that deadlocks in the general case may not deadlock in a restrictive system model.

In the following, we review some concepts of coherence protocols. Note that in this work we focus on directory protocols that enforce the Single-Writer-Multiple-Reader (SWMR) invariant. These are the most common protocols employed today in industry and comprise the MOESIF family of protocols. Our work is applicable to industrial-strength coherence protocol specifications, including CHI [7], CXL [8], and OpenCAPI [13].

A. Directory and Cache Controllers

A directory coherence protocol consists of a set of cache controllers and directory controllers exchanging messages to keep the caches consistent. There can be multiple directory controllers, each in charge of handling a mutually exclusive set of cache block addresses. Formally, these controllers can be modeled as finite state machines (FSMs) that maintain, for every cache block individually, the state of that block: variants of the MOESIF states. These state machines also specify what should happen when messages pertaining to a cache block arrive; typically the state of the cache block changes, and in addition, other messages may be sent to other controllers. These state machines are often specified in a tabular format as in Nagarajan et al. [11]—reproduced in Figures 1 and 2—where the rows contain states and columns contain message ids, and each cell specifies what happens when a message is received in a specific state.

B. Static Messages and Dynamic Instances

We distinguish between a message event, which refers to the message instance during execution, and the message name, which is a static concept. While the columns in protocol tables statically specify the names of messages (ids) that will be received, multiple instances of these messages are received throughout any execution (dynamically). In this paper, we use *message* to refer to the static message name (id), and we use *instances* or *events* to refer to the multiple instances of that message type that occur during a concrete execution (which have concrete values, like addresses, associated with them).

C. Coherence Transactions

A cache coherence transaction is initiated by a cache controller. Each coherence transaction consists of a series of coherence message events, starting with a request and ending in a response. Each of these coherence message events carries a name that encodes the function of the message. A GetS message (GetShared), for example, is a read request that is initiated on a processor load, that is sent from the cache to the directory. A GetM message (GetModified) is a write request that is initiated on a processor store, that is again a message sent from the cache to the directory. In addition to read and write requests, there are also cache block eviction requests. Each of these requests (reads, writes, and evictions) is sent to the home directory that is in charge of that cache block address; the directory responds to the request if it can, or it sends a forwarded request to the cache that is currently the owner of that address; that owner cache then responds to the directory and/or the original requestor. In some protocols, the requesting cache additionally sends a completion message to the directory, signaling the end of the coherence transaction.

It is useful to classify coherence message names into types: requests (which always go from the cache to the directory), forwarded requests (from the directory to the cache), and responses, which can be further classified into data responses and control responses. Summarizing, each message instance has a name, and each message name has a type (request, forwarded request, data response, and control response).

D. Interconnect Network Model

The cache and the directory controllers communicate with each other across an interconnect network (ICN). Each controller has one or more outbound and inbound FIFO queues which buffer outgoing and incoming coherence messages. (FIFO queues are predominantly used because of their simplicity.) The ICN could provide point-to-point ordering or no ordering. These FIFO queues are also known as virtual networks (VNs), because each FIFO queue simulates a separate network connecting the caches and the directory.

E. Protocol Stalls

The correctness of coherence protocols hinges on each coherence transaction appearing to take place atomically. However, modern ICNs and protocols are highly concurrent, and so multiple different coherence transactions can be in flight at any given time. To ensure logical atomicity, a cache or directory controller may choose to delay the processing of a message of the same address as that of the transaction that is currently in flight. This is typically done by blocking the incoming FIFO queue. Going back to the protocol in Figure 2, a directory controller that is currently in a transient state—having received a GetS to address X and having forwarded the request to the current owner—blocks another GetM to the same address until the first transaction has been completed. In other words, the directory blocks the second GetM until it receives a data response for the GetS This delay is represented in the protocol as a *stall*—the semantics of which is to block the incoming queue until the original transaction completes.

III. The Failure of Conventional Wisdom

As discussed in Section I, conventional wisdom says that the number of VNs depends on the longest chain of message dependencies. For many directory protocols that chain length is three: request, forwarded request, response. Some protocols, which follow a response with a completion message from the requestor to the directory, have a chain length of four. In this section, we show that this analysis is neither necessary nor sufficient.

A. Not Sufficient

To demonstrate that conventional wisdom can lead to a number of VNs that is not sufficient, we show an example of a protocol for which conventional wisdom would require three VNs. Start with the standard MSI directory protocol in Table 8.1 of Nagarajan et al.'s Primer on Memory Consistency and Cache Coherence [11]—reproduced as Figures 1 and 2. Assume a system with 3 caches and 2 directories. One of the directories is the home of block X (denoted Dir-X), and the other is the home of block Y (Dir-Y).

Initially, cache C1 holds block X in state M, and cache C2 holds block Y in state M. Cache C3 holds neither block. Dir-X records that X is in state M with C1 as the owner, and Dir-Y records that Y is in state M with C2 as the owner.

The example execution proceeds in four time steps, each of which consists of two concurrent actions. We illustrate the example in Figure 3 and explain it below.

- Time 1
  - C1 sends a GetM request for Y to Dir-Y and transitions to a transient state (denoted IM^{AD} in Figure 1). Dir-Y receives the request and changes state to M with C1 as owner. Dir-Y sends a Fwd-GetM to C2 that is delayed until time 4.
  - C2 sends a GetM request for X to Dir-X and transitions to a transient state. Dir-X receives the request and changes state to M with C2 as new owner. Dir-X sends a Fwd-GetM to C1 that is delayed until time 4.
- Time 2
  - C3 sends a GetM for Y to Dir-Y. Dir-Y changes state to M with C3 as owner. Dir-Y sends Fwd-GetM to C1.
  - C3 sends a GetM for X to Dir-X. Dir-X changes state to M with C4 as owner. Dir-X sends Fwd-GetM to C2.
- Time 3
  - C1 receives Fwd-GetM for Y, but stalls because it is in a transient state.
  - C2 receives Fwd-GetM for X, but stalls because it is in a transient state.
- Time 4
  - C1 receives Fwd-GetM for X, but it is stalled behind Fwd-GetM received at time 3.
  - C2 receives Fwd-GetM for Y, but it is stalled behind Fwd-GetM received at time 3.

	Load	**Store**	**Replacement**	**Fwd-GetS**	**Fwd-GetM**	**Inv**	**Put-Ack**	**Data from Dir (ack=0)**	**Data from Dir (ack>0)**	**Data from Owner**	**Inv-Ack**	**Last-Inv-Ack**
I	Send GetS to Dir/IS^{D}	Send GetM to Dir/IM^{AD}										
IS^{D}	Stall	Stall	Stall			Stall		-/S		-/S		
IM^{AD}	Stall	Stall	Stall	Stall	Stall			-/M	-/IM^{A}	-/M	ack−−	
IM^{A}	Stall	Stall	Stall	Stall	Stall						ack−−	-/M
S	Hit	Send GetM to Dir/SM^{AD}	Send PutS to Dir/SI^{A}			Send Inv-Ack to Req/I						
SM^{AD}	Hit	Stall	Stall	Stall	Stall	Send Inv-Ack to Req/IM^{AD}		-/M	-/SM^{A}		ack−−	
SM^{A}	Hit	Stall	Stall	Stall	Stall						ack−−	-/M
M	Hit	Hit	Send PutM +data to Dir/MI^{A}	Send data to Req & Dir/S	Send data to Req/I							
MI^{A}	Stall	Stall	Stall	Send data to Req & Dir/SI^{A}	Send data to Req/II^{A}		-/I					
SI^{A}	Stall	Stall	Stall			Send Inv-Ack to Req/II^{A}	-/I					
II^{A}	Stall	Stall	Stall				-/I					

Fig. 1. MSI Cache Controller from Nagarajan et al. [11]

	GetS	**GetM**	**PutS-NonLast**	**PutS-Last**	**PutM + data from Owner**	**PutM + data from Non-Owner**	**Data**
I	Send data to Req, add Req to Sharers/S	Send data to Req, set Owner to Req/M	Send Put-Ack to Req	Send Put-Ack to Req		Send Put-Ack to Req	
S	Send data to Req, add Req to Sharers	Send data to Req, send Inv to Sharers clear Sharers, set Owner to Req/M	Remove Req from Sharers send Put-Ack to Req	Remove Req from Sharers, send Put-Ack to Req/I		Remove Req from Sharers, send Put-Ack to Req	
M	Send Fwd-GetS to Owner, add Req and Owner to Sharers, clear Owner/ S^{D}	Send Fwd-GetM to Owner, set Owner to Req	Send Put-Ack to Req	Send Put-Ack to Req	Copy data to memory, clear Owner, send Put-Ack to Req/I	Send Put-Ack to Req	
S^{D}	Stall	Stall	Remove Req from Sharers, send Put-Ack to Req	Remove Req from Sharers, send Put-Ack to Req		Remove Req from Sharers, send Put-Ack to Req	Copy data to memory /S

Fig. 2. MSI Directory Controller from Nagarajan et al. [11]

The underlying reason for the deadlock is that a Fwd-GetM for one cache block is stalling and a Fwd-GetM for another cache block is stuck behind it. The cache cannot process the first Fwd-GetM until it has processed the second.

To avoid deadlock, these two messages would need to be on separate virtual networks, but they are both the same type of message. One way to separate them would be to provide a separate set of virtual networks for every cache block, which is not practical.

We have found multiple examples like this for well-known and seemingly standard protocols [11]. We hypothesize that they may have been designed for one directory, and might not have been verified or tested with multiple directories.

B. *Not Necessary*

We now present two examples in which the number of VNs is *less* than what conventional wisdom would require.

The first example is almost trivial. Imagine a directory protocol in which no messages ever block, at either caches or directories.[2] In such a protocol, one does not even need VNs! There is no need to separate messages, because messages can never get stuck behind each other. Nevertheless, conventional wisdom would have determined the need for three or four VNs, depending on the chain of dependent message types.

[2] A completely non-blocking directory generally requires the presence of an O(wned) state (e.g., an MOSI or MOESI protocol) and a directory with enough state to track the maximum number of possible outstanding requests to main memory.

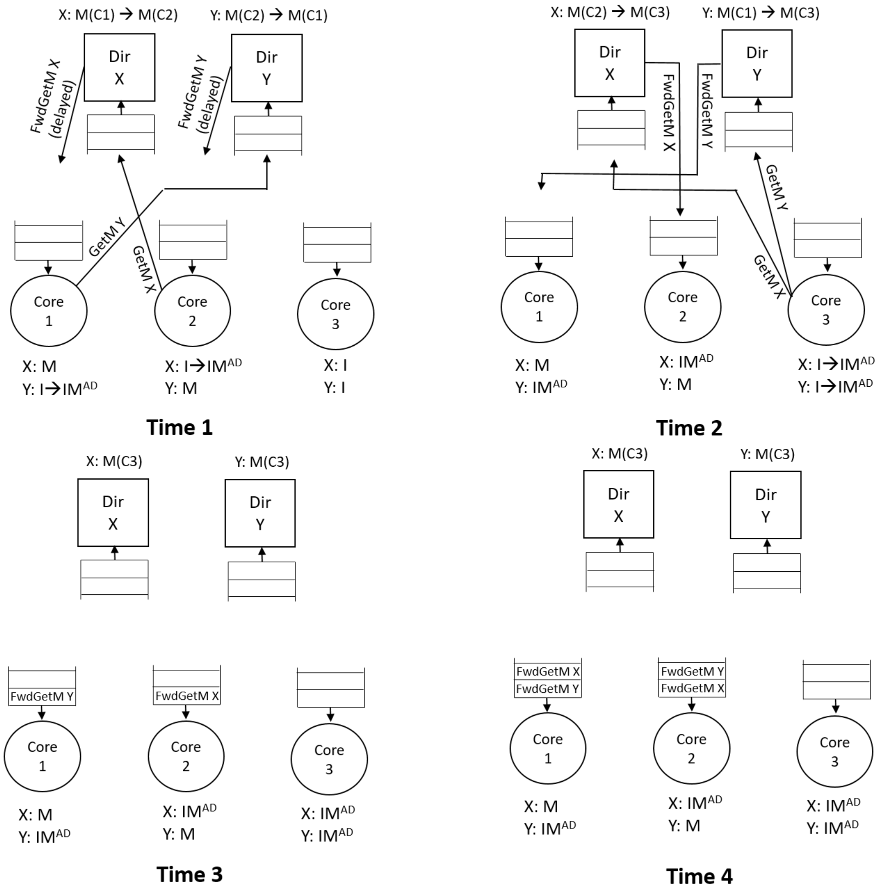

Fig. 3. Deadlock Example

The second example is the industrial CHI protocol [7]. The CHI specification includes four VNs. Our analysis, shown later, reveals that CHI needs only two VNs.

IV. Formalism for Statically Analyzing Coherence Protocols

The main claims of this paper all pertain to the existence of deadlocks in cache coherence protocols. To be able to reason about protocols and detect deadlocks, we develop a formalism for *statically* analyzing coherence protocols and ensuring the absence of deadlocks.

A. Modeling Coherence Transactions

We model coherence transactions as follows: We define a relation $\xrightarrow{\text{causes}} \subseteq M \times M$ on pairs of message names, where $m_1 \xrightarrow{\text{causes}} m_2$ iff the message named m_1 can appear before a message named m_2 in a coherence transaction. Note that the relation above is static; this means that there exists a coherence protocol execution in which the message named m_1 comes before m_2 in a coherence transaction, even if this is not the case for every protocol execution. The class of coherence protocols we look at—the MOESIF family of protocols—involves a rich set of coherence transactions. In the MSI protocol in Figures 1 and 2, some coherence transactions can be as simple as:

$$\mathsf{GetS} \xrightarrow{\text{causes}} \mathsf{Data} \tag{1}$$

A GetS (GetShared) request from a cache causes a data response from the directory itself if the block is owned by the directory. More complex transactions might take the following form:

$$\mathsf{GetS} \xrightarrow{\text{causes}} \mathsf{Fwd\text{-}GetS} \xrightarrow{\text{causes}} \mathsf{Data} \tag{2}$$

In this case, a GetS request from cache-1 leads to a forwarded request being sent to another cache (cache-2, the current owner), which then responds back to the directory and the requestor with the data response (Data). Even more complex coherence transactions are possible. Across every MOESIF protocol, however, the longest chain is at least as long as:

$$\mathsf{req} \xrightarrow{\text{causes}} \mathsf{fwd} \xrightarrow{\text{causes}} \mathsf{res}$$

i.e., a message of type request from a cache causing a message of type forwarded request being sent from the directory to another cache, leading to a message of type response from that cache to the requesting cache.

B. Computing the "causes" Relation

Computing $\xrightarrow{\text{causes}}$ is straightforward, given the protocol state transition tables for the cache and directory controllers (e.g., Figures 1 and 2). The basic idea is to start from a state in the cache and perform a depth-first traversal of the sequence of messages in response to each processor core request (load, store, and eviction). Note that this is a static analysis of the protocol. During the traversal, when a message is sent to a controller, we again trace the sequence of messages for every state that the controller could be in. Every sequence thus starts with a request from a cache and ends in a response. All of these sequences together thus define the $\xrightarrow{\text{causes}}$ relation. Going back to the protocol in Figures 1 and 2, tracing the $\xrightarrow{\text{causes}}$ relation for a read yields equations 1 and 2. Applying a similar process for the write and the eviction yields the complete $\xrightarrow{\text{causes}}$ relation for the protocol.

C. Modeling Protocol Stalls

We model the fact that a message name can stall another message name via a relation called $\xrightarrow{\text{stalls}} \subseteq M \times M$, where $m_0 \xrightarrow{\text{stalls}} m_1$ if, and only if:

- there exists a controller which accepts a message named m_1 in some state, say S, and
- after receiving a message named m_0 and transitioning to a transient state, say T, the controller stalls a message named m_1 (to the same address as that of m_0).

Essentially, it means that m_1 can be stalled by a controller after receiving m_0. In the example discussed earlier in Section II-E, $\mathsf{GetS} \xrightarrow{\text{stalls}} \mathsf{GetM}$ as the GetM is stalled because of a prior GetS message. (Looking at the protocol in Figure 2, we can see that the outstanding GetS can also stall a new GetS request.)

How can a message m_1 that is stalled ever get un-stalled? We know that when m_1 is stalled, it is because of a coherence transaction (say t_0) involving m_0, i.e. $m_0 \xrightarrow{\text{stalls}} m_1$. Thus, m_1 will stall until the coherence transaction t_0 completes.[3] In other words, a protocol stall induces a dependency in which a coherence message from one transaction—the one that is being stalled—waits for one or more messages from the transaction that caused it to stall.

What precisely are these messages that m_1 waits for? All messages m belonging to the transaction t_0 that come after message m_0 in $\xrightarrow{\text{causes}}$:

$$M_2 = \{m \mid m_0 \xrightarrow{\text{causes}}{}^{+} m\}$$

Here, the notation $\xrightarrow{\text{causes}}{}^{+}$ denotes the transitive closure of $\xrightarrow{\text{causes}}$. We can thus model this dependency between messages via a relation called "waits for", $\xrightarrow{\text{waits}} \subseteq M \times M$, where

$$\forall m_2 \in M_2,\ m_1 \xrightarrow{\text{waits}} m_2.$$

This means that there is an execution in which message m_1 from transaction t_1 (to cache block address x) is stalled, waiting for the set of messages M_2 belonging to another transaction t_0 (also to address x).

Summarizing, suppose a message m_0 causes message m_1 to be stalled; then m_1 waits for the set of messages that follow m_0 in the $\xrightarrow{\text{causes}}$ relation. This yields the equation:

$$\xrightarrow{\text{waits}} = \xrightarrow{\text{stalls}}{}^{-1} ; \xrightarrow{\text{causes}}{}^{+} \tag{3}$$

Here, $\xrightarrow{\text{stalls}}{}^{-1}$ means the inverse of the relation $\xrightarrow{\text{stalls}}$ and $(;)$ means the composition of the relations.

Going back to our previous example (Figure 2), recall that the directory controller receives a GetM when it is in the transient state S^D. In this state, it (a) has processed a GetS from an earlier transaction, (b) has sent a Fwd-GetS to the owner and (c) is waiting for a Data response. Note that GetM is made to stall until the data response is received, which in turn requires the Fwd-GetS message to have been received by the owner. Thus:

$$\mathsf{GetM} \xrightarrow{\text{waits}} \mathsf{Fwd\text{-}GetS}, \quad \mathsf{GetM} \xrightarrow{\text{waits}} \mathsf{Data}$$

Note again that $\xrightarrow{\text{waits}}$ is a static relation.

D. Computing the "waits for" Relation

As we can see from Equation 3, the $\xrightarrow{\text{waits}}$ relation can be computed simply from the $\xrightarrow{\text{causes}}$ and $\xrightarrow{\text{stalls}}$ relations. We have seen how to compute $\xrightarrow{\text{causes}}$, so we just need to compute $\xrightarrow{\text{stalls}}$. This, we can obtain from the protocol state transition table: We first identify all of the stalls in the protocol. For each stall, it is trivial to identify the message (m_2) that is stalled. Suppose the stall happens in a transient state; We can go back to see what message (m_1) caused it to be sent to the transient state. From this we can infer that $m_1 \xrightarrow{\text{stalls}} m_2$.

E. Modeling ICN Queuing

When a message named m is stalled by a controller, it blocks an input queue: it blocks the VN to which m is mapped. This has the effect of blocking every other message mapped to the same VN that could be queued behind m. More concretely, suppose a message named m_1 mapped to some VN could be stalled, and suppose there is another message m_2 mapped to the same VN, which could queue behind m_1; we model this situation via a relation called queues behind, $\xrightarrow{\text{queues}} \subseteq M \times M$, where:

$$m_2 \xrightarrow{\text{queues}} m_1$$

Note that, in the general case, these two messages m_1 and m_2 could belong to transactions of different addresses.

How does one compute $\xrightarrow{\text{queues}}$? In this work, we do not want to assume any specific ICN properties. (Indeed, industrial coherence protocol specifications such as CHI or CXL do not make any ICN assumptions either.) Thus, we are forced to make the conservative assumption that any message allocated to the same VN as the stalling message can queue behind it.

[3] It actually will only stall until that controller has finished with the transaction. There might be more messages (e.g. an acknowledgment) that are part of the transaction but received by a different controller. We also consider those here in an over-approximation, but this is not a problem, as this just makes our analysis more conservative.

V. Preventing Deadlocks

With our formal vocabulary for coherence protocols, we are now in a position to analyze the conditions that can lead to a deadlock.

When does a deadlock occur? A deadlock occurs in a protocol execution when there is a circular dependency between message events across different coherence transactions: i.e., message events (dynamically) waiting for one another resulting in a circular dependency.

Recall from the previous section that protocol stalls are the primary reason for these waits. Without a stall in the protocol there can be no deadlocks. There is a secondary reason, and that is when a message gets queued behind other stalled messages.

Before we derive the precise condition for a deadlock, we first classify deadlocks into two kinds: *protocol deadlocks* and *VN deadlocks*. In a protocol deadlock, a circular dependency between messages is caused entirely due to protocols stalls. In contrast, in VN deadlocks at least some of these dependencies are because of messages queued behind one another in a VN.

A. Protocol Deadlocks

Some protocols might have inherent dependencies between messages that result in a deadlock regardless of the number of VNs. The problem is not with the number of VNs or their mappings but with the protocol itself. These protocols have a *protocol deadlock*, and they are the Class 1 protocols mentioned in Section I.

Refer back to our example protocol shown in Figures 1 and 2, and consider the following protocol execution. Assume that there are two caches — Cache 1 and Cache 2 — both of which are caching a particular block in state S. Suppose that both of them want to write to the block; hence they each send a GetM-e to the directory and transition to state $\mathrm{SM}^{\mathrm{AD}}$. (We use an 'e' at the end of the message name to indicate that we are referring to the dynamic message event, as opposed to the static message name.) Assume that the GetM-e from Cache 1 reaches the directory first; accordingly, the directory sends an Inv-e message event to Cache 2 and moves to state M. (Because the GetM-e from Cache 2 reaches the directory second, and the directory is in state M with Cache 1 as the owner, the directory sends a Fwd-GetM-e to Cache 1.)

Referring to Figure 1, the Inv-e would reach Cache 2 in state $\mathrm{SM}^{\mathrm{AD}}$. As per the cache controller specification, the message is served and an Inv-Ack-e message event is sent back to the requestor, which is Cache 1. Suppose that the protocol *instead* stalls the incoming Inv-e message on $\mathrm{SM}^{\mathrm{AD}}$. In that case, the Inv-e message event would wait for a response for the ongoing write that has been initiated at Cache 2; in particular, it would wait for the Data-e response from Cache 1, which in turn means that it would wait for Cache 1 to process the Fwd-GetM-e from the directory. Thus in this execution: $\mathsf{Inv\text{-}e} \xrightarrow{\text{dyn-waiting}} \mathsf{Fwd\text{-}GetM\text{-}e}$. (Note here that $\xrightarrow{\text{dyn-waiting}}$ is a dynamic relation on message events with the first message event waiting for the second in the execution.) But when the Fwd-GetM-e reaches Cache 1, Cache 1 would be in state SM^{A} and would stall the Fwd-GetM-e until it receives the Inv-Ack-e from Cache 2. Thus, in this execution: $\mathsf{Fwd\text{-}GetM\text{-}e} \xrightarrow{\text{dyn-waiting}} \mathsf{Inv\text{-}e}$, leading to a cycle.

Generalizing, a protocol deadlock means that there is an execution with a cycle in $\xrightarrow{\text{dyn-waiting}}$, and none of these dependencies in the cycles are due to messages being queued behind one another in a VN. Such protocols are easily identified using a model checker. We model check the protocol with just one address[4] and with every message allocated its own VN; if there is a deadlock, then that indicates a protocol deadlock. In this work and in the rest of the discussion, we assume that the given protocol is correct in the sense that it does not suffer from a protocol deadlock. Tools such as ProtoGen [14] can be used to ensure that there are no protocol deadlocks.

Our focus is on ensuring that a given protocol (without any protocol deadlocks), will not experience any deadlocks due to VNs and their mappings to message names.

B. VN Deadlocks

A deadlock may arise because of non-stalling messages getting queued behind other stalling messages. These deadlocks arise not because of a protocol bug, but because two messages happened to be allocated to the same VN.

Consider the directory controller in Figure 2. We saw earlier that a GetM can get stalled behind an ongoing GetS transaction. When the data response for the GetS arrives it can get queued behind the stalling GetM leading to the following cycle: $\mathsf{GetM} \xrightarrow{\text{waits}} \mathsf{Data} \xrightarrow{\text{queues}} \mathsf{GetM}$.

Having multiple virtual networks is a way to avoid these deadlocks. A message that was originally queued behind a stalling message—if it was allocated to a different virtual network as that of the stalling message—would now not be queued behind and can progress. In the above example, if requests (cf. GetM) and responses (cf. Data) are assigned to different VNs, this deadlock can be avoided. This is why coherence protocol specifications prescribe multiple virtual networks and allocate message names to specific virtual networks to avoid VN deadlocks.

Generalizing, we see that the interaction of stalled messages waiting for other messages, and stalled messages blocking other messages could lead to a deadlock. We call this deadlock a VN deadlock.

C. Sufficient Condition to Prevent VN Deadlocks

In order to ensure that there are no VN deadlocks, we need to have enough VNs so that there is no cycle in the combination of $\xrightarrow{\text{waits}}$ and $\xrightarrow{\text{queues}}$. What sort of combination? Not a simple union, but a union where there is at least one $\xrightarrow{\text{waits}}$ edge. Why? As we saw earlier, without a stall (which

[4] One address is sufficient for protocol deadlocks as these deadlocks are about protocol bugs and not about one coherence transaction blocked in a VN behind another transaction for a different address.

would manifest as a $\xrightarrow{\text{waits}}$ edge), there can be no deadlocks. More precisely:

$$\text{acyclic}(\xrightarrow{\text{waits}}; (\xrightarrow{\text{waits}} \bigcup \xrightarrow{\text{queues}})^*) \tag{4}$$

Here, $(\xrightarrow{\text{waits}} \bigcup \xrightarrow{\text{queues}})^*$ denotes the reflexive, transitive closure of the union of $\xrightarrow{\text{waits}}$ and $\xrightarrow{\text{queues}}$. We assert that Equation 4 is sufficient to prevent deadlocks: i.e., if the equation above is acyclic then there can be no deadlocks.

To see why, let us suppose there is a deadlock; we will then show that Equation 4 cannot hold. Because we have assumed there is a deadlock, by definition, there has to be a controller (say C_1) whose state machine cannot transition to any other state. This means that the controller C_1 must be stalled in a (transient) state T. (Otherwise, it would be able to transition to a different state.) Let m_1 be the message that is stalled in T. Since C_1 is stalled, it is waiting for a message m_2 (with $m_1 \xrightarrow{\text{waits}} m_2$) which must come from some other controller (say C_2). Since we have assumed there is a deadlock, at this point, the transition that makes C_2 send message m_2 cannot happen. This must be because C_2 is either waiting for a different message (say m_3) or because message m_2 is queued behind a different message (say m_4). We can thus successively build a chain of messages that must, inevitably, end in a cycle because there is a finite number of controllers and messages. The presence of a cycle though implies that Equation 4 does not hold.

D. Preventing Deadlocks

Equation 4 reveals two ways to prevent a deadlock: prevent messages from waiting for one another or prevent messages from being queued behind one another. The root cause of a message waiting for another message is a stall in the protocol. Therefore, avoiding stalls in the protocol is a way to prevent or reduce waiting. We do not explore this approach in this paper, insofar that we do not modify the protocol to prevent or reduce stalling.

The second way to prevent deadlocks is to reduce or prevent messages from being queued behind other messages by allocating messages on to different VNs. This is the focus of our work: we are interested in minimizing the number of VNs necessary to prevent a deadlock. We will develop an algorithm for this in the next section. Before that we discuss the limits of the approach: for certain protocols it is impossible to prevent a deadlock even if we assign each message to a unique VN.

E. Protocols with Inevitable VN Deadlocks

Suppose a protocol has a cycle in the $\xrightarrow{\text{waits}}$ relation. We argue that this protocol will deadlock even if every message name was allocated to its own unique VN. Intuitively, this is because a cycle in $\xrightarrow{\text{waits}}$ allows us to construct a dynamic execution (shown below) where every message queues behind another message of the same name with a different address. Such a cycle can only be prevented if two messages of the same name can be allocated to a different VN, which obviously is impossible. We call such protocols Class 2 protocols.

a) A cycle in $\xrightarrow{\text{waits}}$ implies a VN Deadlock: Suppose that there is a cycle $m_1 \xrightarrow{\text{waits}} m_2 ... m_{n-1} \xrightarrow{\text{waits}} m_n \xrightarrow{\text{waits}} m_1$ in $\xrightarrow{\text{waits}}$, where each m_i is a message name in the protocol. Recall, though, that we have assumed the absence of a protocol deadlock, which means that there exists no corresponding *dynamic* cycle, in $\xrightarrow{\text{dyn-waiting}}$, with messages of those names.

We know, however, that for every individual edge $m_i \xrightarrow{\text{waits}} m_{i+1}$ in the cycle, there exist events $m_i\text{-}e$, $m_{i+1}\text{-}e$ and a protocol execution such that $m_i\text{-}e(a) \xrightarrow{\text{dyn-waiting}} m_{i+1}(a)\text{-}e$, for some address a. In other words, we may not have a chain of $\xrightarrow{\text{dyn-waiting}}$ relating all message events to the same address, but we are guaranteed to have $\xrightarrow{\text{dyn-waiting}}$ edges relating pairs of message events for each pair in the cycle.

Crucially, the state machines for different addresses are independent, so we can combine these pairs of events using *different addresses* to construct a cycle. We do this by chaining these pairs of events with $\xrightarrow{\text{queues}}$ edges, as follows: $m_1\text{-}e(a) \xrightarrow{\text{dyn-waiting}} m_2(a)\text{-}e \xrightarrow{\text{queues}} m_2\text{-}e(b) \xrightarrow{\text{dyn-waiting}} m_3(b)\text{-}e, \ldots, m_{n-1}(c)\text{-}e \xrightarrow{\text{dyn-waiting}} m_n\text{-}e(c) \xrightarrow{\text{queues}} m_n\text{-}e(d) \xrightarrow{\text{dyn-waiting}} m_1\text{-}e(d) \xrightarrow{\text{queues}} m_1\text{-}e(a)$.

Note that each of the $\xrightarrow{\text{queues}}$ edges in the cycle above relates messages of the same name. Therefore, it would be impossible to break the cycle by assigning different messages to different VNs.

b) Explaining the Deadlock in Section III-A: We can now explain why the protocol presented in Figures 1 and 2 suffered from this issue. The cache controller stalls a Fwd-GetM and that Fwd-GetM could potentially wait for another Fwd-GetM. In other words there is a cycle in $\xrightarrow{\text{waits}}$: Fwd-GetM $\xrightarrow{\text{waits}}$ Fwd-GetM. To prevent this cycle, we would need to somehow distinguish these messages with the same name and assign them to different VNs; for example it is theoretically possible to allocate a distinct VN for every cache block address, but that would be impractical.

VI. Practical Protocols Need Two VNs

The key claim of this paper is that practical protocols (neither Class 1, which suffer from protocol deadlocks, nor Class 2, which suffer from inevitable VN deadlocks) need only two virtual networks (VNs) to prevent deadlocks. We use our formalism to derive an algorithm to assign virtual networks such that no deadlocks can occur. This allows us to substantiate this claim.

A. Algorithm for Assigning Virtual Networks

In this section we develop an algorithm for determining the number of VNs necessary to avoid deadlocks. The core idea is that we use Equation 4 to statically compute the cycle(s) that can potentially cause a deadlock (if any). For this initial computation, we assume one VN, which means each message can potentially queue behind any other message. We then minimally partition the messages of the protocol so that we break cycles in Equation 4. Partitioning two messages is tantamount to assigning those to two different VNs. To do this, we reduce the problem of minimizing the number of VNs to a known problem from graph theory: that of computing a *minimum feedback arc set* [15]. In the following, we explain this reduction and why it is correct.

a) Constructing the Graph: To reduce the assignment of VNs to a graph problem, we must first construct an appropriate graph. Let P be a protocol, given as state-transition tables for the directory and cache controllers. We then define the directed graph $G(P) = (E, V)$ as follows. The set of vertices is given by message names $m \in M$, i.e., $V = M$. For two messages m_1, m_2, we add an edge (m_1, m_2) iff there exist a path from m_1 to m_2 starting with a $\xrightarrow{\text{waits}}$ edge and consisting then of $\xrightarrow{\text{waits}}$ or $\xrightarrow{\text{queues}}$ edges. In other words

$$E = \xrightarrow{\text{waits}}; (\xrightarrow{\text{waits}} \bigcup \xrightarrow{\text{queues}})* \qquad (5)$$

The set of edges of the graph G, therefore, corresponds exactly to the relation of Equation 4. Computing this graph is trivial since we know how to compute $\xrightarrow{\text{waits}}$ and $\xrightarrow{\text{queues}}$ relations (as outlined in the previous section).

We want to minimally remove edges in this new graph G so that it is acyclic; recall, from Equation 4 that acyclicity is necessary to avoid deadlocks. Then, we want to translate these removed edges to their corresponding $\xrightarrow{\text{queues}}$ relations between messages in the protocol; recall that we can only break cycles by removing $\xrightarrow{\text{queues}}$ relations (i.e., assigning the two messages in the relation to diferent VNs). For this, we remember the (minimal) paths that we used to build the edges in G, satisfying Equation 5. Note that there can be multiple minimal paths between any two messages consisting of $\xrightarrow{\text{waits}}$ and $\xrightarrow{\text{queues}}$. Since we assume a single VN at this point, these paths can only be either a single $\xrightarrow{\text{waits}}$ or a $\xrightarrow{\text{waits}}$ followed by a $\xrightarrow{\text{queues}}$ or $\xrightarrow{\text{waits}}$. For each edge e, we remember all the $\xrightarrow{\text{queues}}$ relations we find in all minimal paths; We call this set $qs(e)$.

b) Breaking the Cycles: A feedback arc set S of a graph G is a set of edges in G, such that S has an edge for every cycle in G and removing the edges in S from G makes G acyclic. If S is minimal with this property, it is called a minimal feedback arc set [15]. This can also be computed in a weighted version, where the edges have weights.

From the graph G we computed earlier, we want to remove just the $\xrightarrow{\text{queues}}$ relations to make it acyclic. Thus, we define the weights as follows:

$$w(e) = \begin{cases} 1, & \text{if } \mathrm{qs}(e) \neq \emptyset, \\ 2^{|V|} + 1, & \text{otherwise.} \end{cases} \qquad (6)$$

With this definition, the sum of the weights of all edges e, where the path in Equation 5 contains a $\xrightarrow{\text{queues}}$ edge, is less than the weight of any single edge that consists only of $\xrightarrow{\text{waits}}$ edges. In particular, for an edge e in a minimal feedback arc set S, if $\mathrm{qs}(e) = \emptyset$ (i.e. it consists only of $\xrightarrow{\text{waits}}$ edges), Equation 4 means that $\xrightarrow{\text{waits}}$ is cyclic. (Otherwise, if all the cycles in Equation 4 have $\xrightarrow{\text{queues}}$ edges, there would be a different edge in the minimal arc set S with lower weight.) In this case, when there is a $\xrightarrow{\text{waits}}$-only cycle, it means this protocol is a Class 2 protocol; we give up on the computation, since there is no way to avoid the VN deadlock by assigning virtual networks per message ID. On the other hand, if all cycles have at least one $\xrightarrow{\text{queues}}$ edge, then we can break these cycles with a VN assignment.

c) Finding a VN assignment: Given a minimal feedback set S of our computed graph G, we can use it to compute the minimal number of VNs required. We translate each edge $e \in S$ to all the $\xrightarrow{\text{queues}}$ relations $\mathrm{qs}(e)$ that we previously remembered.

We then use these $\xrightarrow{\text{queues}}$ edges from the minimal feedback arc set S to construct an undirected conflict graph. For each such $m_1 \xrightarrow{\text{queues}} m_2$ we add (m_1, m_2) to the conflict graph. We then find a minimal graph coloring of this conflict graph. This coloring will give a partial partition of the message IDs into VNs that ensures no cycles in Equation 4. This is by construction: the minimal feedback arc set is exactly the set of edges we need to remove to break all cycles. By coloring the conflict graph, we ensure that all messages that could cause such a deadlock are in different VNs. The remaining message ids (that were not part of the conflict graph) can be distributed among the virtual networks in any way, since they cannot cause any VN deadlocks.

Note that the conflict graph cannot have any self edges, since the paths are minimal by construction. Recall that these cycles have to start with a $\xrightarrow{\text{waits}}$ edge, so that removing an edge $m_1 \xrightarrow{\text{queues}} m_1$ would always leave a cycle.

B. Complexity and Tractability

The problem of finding shortest paths to construct the graph is known to be polynomial in the size of the graph, as is the computation of $\xrightarrow{\text{waits}}$ and $\xrightarrow{\text{queues}}$ as we discussed in Section IV. Both graph coloring and the problem of finding a minimum feedback arc set for a directed graph are in Karp's 21 classical NP-hard problems [16]. Many heuristics and algorithms have been studied in practice, which make this tractable in practice [17]–[19]. In particular, for our cases, the instances are fairly small, as the number of nodes in the graph is the number of message IDs in the protocol, and the order of magnitude is around 10^1.

C. Implications of our Theory

In this section, we analyze the implications of our theory and draw useful conclusions for the three classes of protocols we introduced earlier in Section I.

1) Class 1 Protocols: Protocol Deadlock: As discussed in Section V-A, a protocol with a cycle in $\xrightarrow{\text{dyn-waiting}}$ suffers from a protocol deadlock. These Class 1 protocols, which can be identified by model checking (e.g., as has been done with Murphi [12], [20]), are doomed; VNs are not going to save the day.

2) Class 2 Protocols: Stalling Forwarded Messages Considered Harmful: In Section V-A, we showed that a protocol with a cycle in $\xrightarrow{\text{waits}}$ inevitably experiences a VN deadlock even if every message is allocated its own VN.

What kinds of protocols will have a cycle in $\xrightarrow{\text{waits}}$, and how can these be avoided? We provide a qualitative analysis. First, note that in any protocol a response message (data or control response) cannot ever be stalled, as stalling those messages will lead to a protocol deadlock. Second, note that, while a request message can be stalled, stalling requests cannot cause a cycle in $\xrightarrow{\text{waits}}$. This is because a request message can only wait for forwarded requests and responses but can never wait for another request. (Again, doing so will lead to a protocol deadlock.) Thus, we are left with forwarded requests.

We assert that protocols that stall forwarded requests are harmful, as they could potentially lead to a cycle in $\xrightarrow{\text{waits}}$. This is because in a protocol that stalls forwarded messages, it might be possible for a message to wait for a message of the same name. Going back to Section III-A, recall that this was the problem in the protocol (Figure 1): in that protocol, a Fwd-GetM waits for another Fwd-GetM.

It is important to note that there are two situations in which a protocol can stall forwarded requests and yet remain deadlock-free. First, one can contrive odd protocols to avoid cycles. For example, if a protocol were to stall on only one type of forwarded request but not the others, it might be possible to prevent a cycle in $\xrightarrow{\text{waits}}$, and one could put that forwarded request on one VN and the rest on another VN. Second, if one limits the system to one centralized directory that handles requests for all addresses, it is similarly possible to eliminate cycles that could otherwise occur.

3) Class 3 Protocols: Practical Protocols Require 2 VNs: Class 3 protocols have neither protocol deadlock nor do they let caches stall forwarded requests. Class 3 protocols are distinguished by whether the directories stall incoming requests, and there are two possibilities.

First, if the directories never stall, only one VN is needed, because there are no $\xrightarrow{\text{waits}}$ edges in these protocols. While attractive, they are generally not practical due to the need for every directory to have enough MSHRs to track the worst-case number of outstanding requests to off-chip memory.

Second, if the directories sometimes or always stall requests, deadlocks can be avoided by assigning requests to their own VN, while other message types share another VN. Why is this correct? Because requests are the only types of messages that can stall, and because requests could wait for forwarded requests and responses—from Equation 4, it becomes clear that isolating requests from all other messages using two different VNs prevents deadlocks. This second group of Class 3 protocols—with directories that stall at least some requests—are practical, and they all require two VNs. It is worth noting that this class includes industrial protocol specifications including CHI and CXL.

Thus, a tangible practical implication of this work is that each of the protocols in this class need only 2 VNs for correctness whereas their specifications mandate more than 2 VNs—e.g., the CHI specification mandates 4 VNs. This reduced VN requirement has important ramifications for power, performance, and area (PPA) of Network-on-chips (NoCs). The impact is significant: recent work has shown that reducing the number of VNs from 6 to 0 saves 40% NoC power and area, and reduces critical path time by 31% [3]. While our algorithm does not always result in the elimination of VNs—it only results in the elimination of VNs for fully non-stalling protocols—it nevertheless shows that our proposal could translate into non-trivial benefits for PPA. Importantly, these benefits are for "free" because our proposal does not need new microarchitectural mechanisms to tolerate deadlocks. And finally, with cache-coherent interconnect specifications such as CHI and CXL starting to get used widely for integrating accelerators, these benefits can be reaped across the board.

Looking ahead, when new protocol specifications are designed, our analysis provides the minimum VNs needed to avoid deadlocks. This does not mean that the system designer must necessarily pick the exact number prescribed by our analysis; rather, it allows the designer to focus on performance considerations without the burden of having to consider deadlocks. We expect designers to choose the minimum number of VNs, but a designer may choose to use more; for example, a designer might prefer to separate message types of different sizes that our algorithm maps to the same VN.

VII. Verification of Deadlock Freedom

To corroborate our analysis, we have used model checking to verify the absence/presence of deadlocks in a variety of protocols. Because of its suitability for coherence protocol verification, we use the Murphi model checker [12]. We run Murphi on a computer with 256 GB of memory, in order to maximize its ability to handle very large state spaces.

Even with 256 GB, Murphi does not scale to the systems we need to model, if modeled naively. We have innovated how we model systems in the model checker, as discussed later in this section. All of the protocols that deadlock are indeed detected by model checking. Of the protocols that do not deadlock, we completely check some, but some others have state spaces that exhaust 256 GB of memory, even with our optimizations. For these protocols, we use the well-known technique of bounded model checking [21]. With bounded model checking, the model checker uses breadth-first search and progresses level by level until the model checker is halted. All of our bounded model checking experiments have reached at least 47 levels, which is significantly more than the depths at which we detect deadlocks in the protocols that have deadlocks (25-31).

A. Models of Protocols for Model Checking

For any given protocol, we seek to verify the most general system model. This includes an arbitrary network topology and a number of caches, addresses, and directories that is sufficient to manifest any possible deadlocks.

1) Interconnection Network: Directly modeling the interconnection network—including its topology and buffers—for every possible topology is beyond the state

	Directory never blocks MOSI, MOESI	Directory always blocks CHI	Directory sometimes blocks MSI, MESI
Cache never blocks	(1) ✓1 VN	*(3) irrelevant*	(5) ✓2 VN E.g., VN1=Req,VN2={FwdReq, Resp}.
Cache sometimes blocks	(2) deadlocks with 3 VNs	(4) ✓2 VN; E.g., VN1=Req, VN2={Resp,Data,Snoop}	(6) deadlocks with 3 VNs

TABLE I
Summary of Verification Experiments

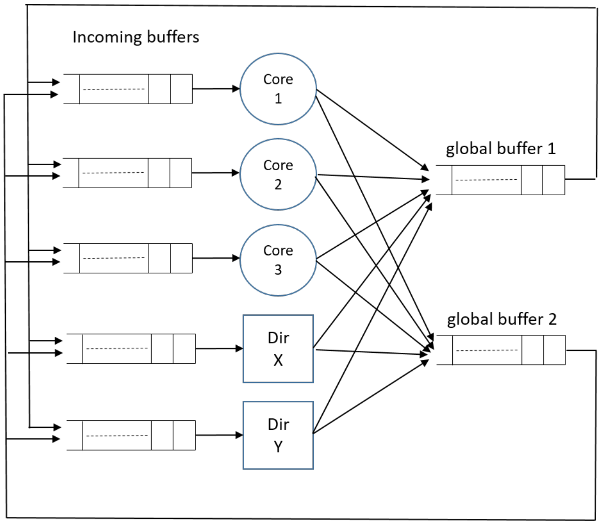

Fig. 4. Model showing only one virtual network, for clarity. (All virtual networks are modeled in the same way.) If the virtual network does not require point-to-point ordering, a source can dynamically choose either global buffer when sending to any destination. If the virtual network does require point-to-point ordering, we separately model check all possible static mappings of source-destination pairs to global buffers.

space capability of a model checker. We instead use a novel construction that preserves all of the possible behaviors of any ICN, including point-to-point ordering and queuing at routers between endpoints. As illustrated in Figure 4, we model each virtual network with (a) a pair of global FIFO buffers and (b) a FIFO buffer at each cache and directory (that can receive messages on that virtual network).[5]

If the virtual network does not require point-to-point ordering, a source can dynamically choose either global buffer when sending to any destination. Because the model checker explores the entire reachable state space, this model enables any pair of messages to either queue behind each other (if inserted into the same global buffer) or not (if inserted into different global buffers). This model also permits arbitrary reorderings of messages in the network, because the model checker will explore situations in which messages in one global buffer stay in that buffer while messages in the other global buffer are popped and delivered to their destinations.

When we wish to preserve point-to-point ordering, we require all messages from Endpoint A to Endpoint B to be inserted into the same global buffer. Thus we have all of the behaviors described above, but with the restriction of point-to-point order. Because deadlocks can depend on message paths, we separately model check $every$ possible static mapping of endpoint-to-endpoint messages to global buffers.

2) Caches, Addresses, and Directories: Practically, the number of caches, addresses, and directories required to manifest coherence bugs, including deadlocks, is limited. To enable the model checker to explore the search space, verification exploits those limits and verifies systems that are no larger than needed to manifest all/most possible deadlocks. Typical verification uses only two or three caches, one or two addresses, and one directory [22], [23]. We have already observed deadlocks that require at least three caches, two addresses, and two directories, and we use those values in our verification experiments.

B. Protocols

In Table I we present our results for a variety of protocol types. The MSI, MESI, and MOSI protocols are inspired by the corresponding protocols in Nagarajan et al. [11]. The MOESI protocol was derived from the MESI and MOSI protocols. For all of these protocols, we modified the cache and directory controllers to add/remove blocking on forwarded requests and requests, respectively. The only completely non-blocking directories are for the MOSI and MOESI protocols; the O(wned) state avoids directory blocking when a cache line transitions from M to S, and we provision enough transient state (MSHRs) at the directory to accommodate the maximum possible number of outstanding requests to main memory.

The CHI protocol is based directly on the CHI prose specification [7]. This is an industrial strength protocol specification that is used widely, and consists of tens of pages of prose specification. This is the first formalization of the CHI protocol to our knowledge. We open source all our Murphi models including the one for the CHI specification.

C. Results

In Table I, we summarize our results, which confirm what we discovered in this paper. In the table, we label the experiments from (1) to (6). Experiments (2) and (6) correspond to Class 2 protocols. Experiments (3), (4), and

[5] This model suffices for any protocol that limits the number of in-flight messages from any node to any other node to two; we are unaware of any protocol that violates this limit.

(5) correspond to Class 3 protocols. Note that experiment (3), with an always blocking directory and a never-blocking cache is irrelevant; the cache will never have an opportunity to take advantage of being non-blocking, because the blocking directory precludes any concurrency.

It is worth noting that protocol (1) which never blocks—either in the cache or the directory controller—requires exactly one VN. Class 3 protocols, which corresponds to experiments (4) and (5), requires exactly two VNs. It is worth noting that this includes the CHI protocol: whereas the specification mandates four VNs, we find that two would suffice. Specifically, the VNs prescribed by CHI are: request, snoop, response, and data response, whereas we have found that mapping the requests to one VN, and the rest of snoops, responses, and data responses to another VN would suffice to avoid deadlocks.

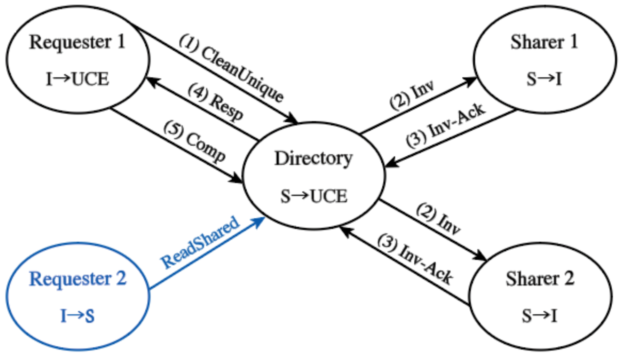

Fig. 5. Example from CHI results. The ReadShared from Requester-2 is waiting for the transaction initiated by Requestor-1 to complete.

To further explain our CHI results, we present the $\xrightarrow{\text{causes}}$ and $\xrightarrow{\text{waits}}$ relations for a small subset of the CHI protocol. The CHI protocol is a MOESI-based intervention-forwarding protocol in which every coherence transaction involves a completion message to the directory. Figure 5 shows Requester-1, which is currently in Invalid state, requesting the directory for write permissions via a CleanUnique request; UCE (Unique Clean Exclusive) is a state that allows for the cache to perform a "full write" to the cache line and so the transaction from I to UCE does not result in data transferred from the directory.[6] A CleanUnique request is sent to the directory which forwards invalidation messages to the sharers; the sharers send acknowledgments to the the directory; once all of the acknowledgments have been received, the directory responds to the requester, which then sends a completion message to the directory.

This leads to the following $\xrightarrow{\text{causes}}$ relation.

$$\begin{aligned} \mathsf{CleanUnique} &\xrightarrow{\text{causes}} \mathsf{Inv} \xrightarrow{\text{causes}} \mathsf{Inv\text{-}Ack} \\ &\xrightarrow{\text{causes}} \mathsf{Resp} \xrightarrow{\text{causes}} \mathsf{Comp} \end{aligned} \tag{7}$$

Now suppose there is another cache which is currently in Invalid state that reads the same cache line and therefore sends a concurrent ReadShared transaction—with the ReadShared ordered at the directory after the CleanUnique. Because the CHI protocol blocks concurrent transactions at the directory, the ReadShared blocks at the directory waiting for the prior CleanUnique transaction to complete. This leads to the following $\xrightarrow{\text{waits}}$ relation:

$$\mathsf{ReadShared} \xrightarrow{\text{waits}} \{\mathsf{Inv},\quad \mathsf{Inv\text{-}Ack},\quad \mathsf{Resp},\quad \mathsf{Comp}\}$$

Generalizing across all coherence transactions, we get the following $\xrightarrow{\text{waits}}$ relation:

$$\mathsf{req} \xrightarrow{\text{waits}} \{\mathsf{fwd},\quad \mathsf{res},\quad \mathsf{data}\}$$

The fact that only requests are ever blocked allows our algorithm to assign requests on one VN and the rest of the messages on the other virtual network.

D. *Model checking times*

Model checking was conducted on a high-performance server: Intel Xeon Gold 6226 Processor with 768 GB memory. Model checking took up to 68 hours and consumed 300 GB memory. Note that these times are in line with other model checking results. Note also that our goal in this paper is not to invent a novel verification approach; our verification results are merely to back up our theory and static algorithm.

VIII. Related Work

The most relevant related work are schemes that address deadlocks in coherence protocols, and not just routing deadlocks. Sorin et al. [6] observed that deadlocks are rare, even with fewer than the required number of VNs, and they proposed deadlock detection (e.g., with a timer) and system recovery. DRAIN [5] similarly notes that deadlock is rare and periodically drains the ICN to recover from any deadlocks that do occur; because deadlock is rare, draining can be infrequent. Both Pitstop [4] and FastPass [3] dedicate certain resources to guarantee that they can unblock packets that are stalled or potentially stalled due to deadlock. Pitstop and FastPass eliminate VNs throughout the ICN, but they appear to require VN buffering at endpoints (caches and directories). More importantly, automatically verifying that such techniques do not suffer from deadlocks is a hard and open problem.

In this work we focus on protocol deadlocks that are caused by inbound FIFO queues that buffer coherence messages at cache and directory controllers. Complex controllers could potentially relax this FIFO ordering to service younger messages before older messages to avoid deadlocks or increase utilization. For example, gem5's Ruby coherence protocols [24] support FIFO-like buffers that allow a blocked message at the head of the FIFO to be removed and placed at its tail. However, using such an approach to completely avoid deadlocks is tricky when point-to-point ordering is required in certain VNs—as is often the case for typical protocols.

[6] For ease of exposition, we have used the standard terminology to represent for other messages in the transaction instead of the CHI terminology.

Other related work focuses strictly on routing deadlocks, rather than deadlocks due to the coherence protocol. Bufferless routing—such as hot potato or deflection routing and BLESS [25]—obviously guarantees deadlock freedom, but implementations must be careful to guarantee eventual forward progress for misrouted packets. Another option is the use of escape virtual channels—in addition to the normally used virtual channels—that are guaranteed to be deadlock-free (e.g., by restricting turns [26]). SEEC [27] combines escape virtual networks with bufferless routing to create stochastic escape express channels. SWAP [28] allows deadlocks to occur and then breaks them by swapping packets in a way that is guaranteed to enable forward progress. SPIN [29] also allows a deadlock to happen; it then orchestrates a *spin* – a coordinated forward movement of every flit in the deadlocked ring. They show that forward progress can be guaranteed with a bounded number of these spins.

There has been a lot of influential work on cache coherence verification, ranging from manual proofs [30] through theorem proving to model checking [20]. Automated model checking techniques have matured enough that there are now standard industrial tools from companies like Cadence that are used widely. None of these tools are immune to state space explosion, however, and practitioners must either automatically verify smaller system models or employ manual intervention [31] to prove for the general case. The primary goal of this paper is *not* to invent a new verification methodology. Rather, our contribution is a theory and an algorithm for deriving a minimal number of VNs for a given protocol. We do use model checking to validate our theory, and we were forced to innovate to manage the state space involved in a system model with more than one directory, which is necessary to trigger the deadlocks.

Our formal approach to modeling coherence protocols bears some resemble to how memory consistency models are modeled axiomatically. Specifically, our approach of statically analyzing coherence protocols to determine the minimal number of VNs is analogous to how a parallel program can be statically analyzed to determine the minimal number of fences that need to be inserted to ensure sequential consistency [32].

IX. Conclusions

Virtual networks play a critical role in preventing deadlock in cache coherence protocols, yet we have discovered that they have not been well understood. Architects relying on conventional wisdom are not guaranteed to find a necessary or sufficient number of VNs. To overcome this problem, we have developed a formalism for describing coherence protocols, how coherence messages can queue in an ICN, and the sufficient condition for determining whether a protocol can deadlock. Using this formalism, we have designed an algorithm that generates the minimum number of VNs required to avoid deadlock, as well as the mappings from message types to VNs. We show that some textbook protocols require an absurd number of VNs, and we show that the CHI protocol was provisioned with more VNs than necessary.

X. Acknowledgments

This material is based on work supported by the National Science Foundation under grant CCF-200-2737. We also acknowledge funding from EPSRC grants EP/V028154/1 and EP/V038699/1 to the University of Edinburgh, and support from the University of Utah. We thank the anonymous reviewers and David Wood for feedback on the paper.

References

[1] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, "The Stanford DASH multiprocessor," *IEEE Computer*, vol. 25, no. 3, pp. 63–79, Mar. 1992.

[2] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The stanford flash multiprocessor," in *Proceedings of the 21st International Symposium on Computer Architecture*, 1994, pp. 302–313.

[3] H. Farrokhbakht, P. V. Gratz, T. Krishna, J. San Miguel, and N. Enright Jerger, "Stay in your lane: A NoC with low-overhead multi-packet bypassing," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, 2022.

[4] H. Farrokhbakht, H. Kao, K. Hasan, P. V. Gratz, T. Krishna, J. San Miguel, and N. Enright Jerger, "Pitstop: Enabling a virtual network free network-on-chip," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, 2021.

[5] M. Parasar, H. Farrokhbakht, N. Enright Jerger, P. V. Gratz, T. Krishna, and J. San Miguel, "Drain: Deadlock removal for arbitrary irregular networks," in *Proceedings of the International Symposium on High-Performance Computer Archtitecture*, 2020.

[6] D. J. Sorin, M. M. Martin, M. D. Hill, and D. A. Wood, "Using speculation to simplify multiprocessor design," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004.

[7] "The AMBA CHI Specification," https://developer.arm.com/architectures/system-architectures/amba/amba-5, accessed: 15th July 2019.

[8] "Compute Express Link," https://www.computeexpresslink.org/, accessed: 18th June 2021.

[9] H. Cook, "Productive design of extensible on-chip memory hierarchies," Ph.D. dissertation, University of California, Berkeley, 2016.

[10] H. Farrokhbakht, H. Kao, and N. D. Enright Jerger, "Ubernoc: unified buffer power-efficient router for network-on-chip," in *Proceedings of the 13th IEEE/ACM International Symposium on Networks-on-Chip, NOCS*, 2019, pp. 1:1–1:8.

[11] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, 2nd ed., ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[12] D. L. Dill, "The Murphi Verification System," in *Proceedings of the 8th International Conference on Computer Aided Verification*, 1996, pp. 390–393.

[13] "The OpenCAPI Consortium," https://opencapi.org/, accessed: 21st January 2019.

[14] N. Oswald, V. Nagarajan, and D. J. Sorin, "ProtoGen: Automatically generating directory cache coherence protocols from atomic specifications," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018, pp. 247–260.

[15] D. Younger, "Minimum feedback arc sets for a directed graph," *IEEE Transactions on Circuit Theory*, vol. 10, no. 2, pp. 238–245, 1963.

[16] R. M. Karp, "Reducibility among combinatorial problems," in *Proceedings of a Symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, ser. The IBM Research Symposia Series, R. E. Miller and J. W. Thatcher, Eds. Plenum Press, New York, 1972, pp. 85–103. [Online]. Available: https://doi.org/10.1007/978-1-4684-2001-2_9

[17] G. Even, J. Naor, B. Schieber, and M. Sudan, "Approximating minimum feedback sets and multicuts in directed graphs," *Algorithmica*, vol. 20, no. 2, pp. 151–174, 1998. [Online]. Available: https://doi.org/10.1007/PL00009191

[18] P. Eades, X. Lin, and W. F. Smyth, “A fast and effective heuristic for the feedback arc set problem,” *Inf. Process. Lett.*, vol. 47, no. 6, pp. 319–323, 1993. [Online]. Available: https://doi.org/10.1016/0020-0190(93)90079-O

[19] A. Baharev, H. Schichl, A. Neumaier, and T. Achterberg, “An exact method for the minimum feedback arc set problem,” *ACM J. Exp. Algorithmics*, vol. 26, apr 2021. [Online]. Available: https://doi.org/10.1145/3446429

[20] 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 Computer & Processors, ICCD ’92, Cambridge, MA, USA, October 11-14, 1992*. IEEE Computer Society, 1992, pp. 522–525. [Online]. Available: https://doi.org/10.1109/ICCD.1992.276232

[21] E. Clarke, D. Ouakine, and J. Strichman, “Completeness and complexity of bounded model checking,” in *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2004.

[22] N. Oswald, V. Nagarajan, D. J. Sorin, V. Gavrielatos, T. Olausson, and R. Carr, “Heterogen: Automatic synthesis of heterogeneous cache coherence protocols,” in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, 2020.

[23] R. Komuravelli, S. V. Adve, and C.-T. Chou, “Revisiting the Complexity of Hardware Cache Coherence and Some Implications,” *ACM TACO*, vol. 11, no. 4, 2014.

[24] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[25] T. Moscibroda and O. Mutlu, “A case for bufferless routing in on-chip networks,” in *Proceedings of the International Symposium on Computer Architecture*, 2009.

[26] C. Glass and L. Ni, “The turn model for adaptive routing,” in *Proceedings of the International Symposium on Computer Architecture*, 1992.

[27] M. Parasar, N. Enright Jerger, P. V. Gratz, J. San Miguel, and T. Krishna, “SEEC: Stochastic escape express channel,” in *SC ’21: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[28] ——, “SWAP: Synchronized weaving of adjacent packets for network deadlock resolution,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[29] A. Ramrakhyani, P. V. Gratz, and T. Krishna, “Synchronized progress in interconnection networks (SPIN): A new theory for deadlock freedom,” in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.

[30] S. Park and D. L. Dill, “Verification of FLASH cache coherence protocol by aggregation of distributed transactions,” in *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA ’96, Padua, Italy, June 24-26, 1996*, G. E. Blelloch, Ed. ACM, 1996, pp. 288–296. [Online]. Available: https://doi.org/10.1145/237502.237573

[31] C. Chou, P. K. Mannava, and S. Park, “A simple method for parameterized verification of cache coherence protocols,” in *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, ser. Lecture Notes in Computer Science, A. J. Hu and A. K. Martin, Eds., vol. 3312. Springer, 2004, pp. 382–398. [Online]. Available: https://doi.org/10.1007/978-3-540-30494-4_27

[32] D. E. Shasha and M. Snir, “Efficient and correct execution of parallel programs that share memory,” *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 2, pp. 282–312, 1988. [Online]. Available: https://doi.org/10.1145/42190.42277

Appendix

A. Abstract

This appendix describes the artifact that accompanies this paper. It includes Python code for determining the minimum number of VNs for a given protocol, as well as generating mappings from message types to VNs. It also includes all evaluated protocols in Murphi, corresponding to Experiments (2), (4), (5), and (6) in Table I.[7] We provide scripts to run the algorithm and the model checking for all protocols, either individually or all together.

B. Artifact Check-list (Meta-information)

- **Program:** Our program (in Python) that determines the minimum number of VNs and generates the mappings from message types to VNs.
- **Protocols**. Our protocol models (in Murphi).
- **Run-time environment:** Linux (Ubuntu 20.04), Python 3.8, CMurphi 5.4.9.1
- **Hardware:** System with about 200GB of RAM and Intel Xeon Gold 6226 Processor.
- **Output:** Algorithm generates virtual network assignment for protocols. Model checking verifies the correctness of the assignments.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT

C. Description

1) How to access: Code available on: https://github.com/Author1-isca24/ISCA24-AE

(For blind review, we keep the code author anonymous)

2) Hardware dependencies: The VN assignment algorithm does not require any special hardware. Model checking requires at least 200 GB of RAM to run most tests. Some model checking runs will require several days to run, for reasons we explain in section F.2.

3) Software dependencies:

- Linux distribution (e.g. Ubuntu 20.04) or MacOS (version newer than 10.15.7)
- Python 3.8 or higher
- networkx 2.6.3 (Python package)
- matplotlib 3.4.3 (Python package)
- CMurphi 5.4.9.1

D. Installation

1) Option 1: Install manually:

1. CMurphi setup

To install CMuprhi run from the parent directory; get CMurphi 5.4.9.1 by: `git clone https://github.com/Errare-humanum-est /CMurphi.git`

Then, install it by:

```
sudo apt-get install bison
sudo apt-get install byacc
sudo apt-get install flex
cd src && make
mkdir ErrorTrace
```

2) Option 2: Use Docker: We have prepared a Docker container with all of the environment (including hardware/software dependencies and CMurphi) already set up. You can access it from: https://zenodo.org/records/10895869

After entering the container, you can start from section E.

E. Using Artifact to Generate VN Results

Since we run the algorithm for several protocols, we provide two options: (a) run all experiments and (b) run a single experiment in Table I.

Run algorithm for all protocols (Run-all option):

```
./run_all_algorithm.sh
```

Run algorithm for single protocol (Run-single option):

For Experiment (2):

```
python3 main.py MOSI
python3 main.py MOESI
```

For Experiment (4):

```
python3 main.py CHI
```

For Experiment (5) :

```
python3 main.py MSI_nonblocking_cache
python3 main.py MESI_nonblocking_cache
```

For Experiment (6) :

```
python3 main.py MSI_blocking_cache
python3 main.py MESI_blocking_cache
```

F. Expected VN Results

Our algorithm results should reproduce Table I.

For the "Run-single" option, when the user runs the algorithm for Experiment (2) and Experiment (6), it will stop because they are Class 2 protocols and output "The protocol is a Class 2 protocol, Program Exit!". For Experiment (4) and Experiment (5), it will generate the possible virtual network assignments for two virtual networks.

For the "Run-all" option, the algorithm is run for all the experiments and it outputs the summary of all the results in Table I.

Since the output text is not very long, it is simply displayed on the command line.

G. Verifying Results

To verify the correctness of the algorithm, we develop Murphi-based protocol models that can be model-checked with Murphi. We again provide two options: (a) model check all experimental results and (b) model check a single experiment from Table I.

Run-all model-checking:

```
./run_all_murphi.sh
```

Run-single model-checking: We provide a command-line input for Experiments (4) and (5) to let the user define the maximum runtime for each sub-task (explained in the following note). Because Experiments (2) and (6) lead to deadlock, they complete quickly (20-40 minutes) and thus do not need an input for maximum runtime. Note that maximum runtime is specified in units of days.

For Experiment (2):

[7] Protocols in categories (1) and (3) of Table I do not need to be evaluated.

```
./run_single_murphi.sh 2
```

For Experiment (4): Since Experiment (4) has 6 sub-tasks, we provide 6 command-line inputs (d1-d6) to set the maximum runtime for each subtask.

```
./run_single_murphi.sh 4 d1 d2 d3 d4 d5 d6
```

For Experiment (5): Since Experiment (5) has 12 sub-tasks, we provide 12 command-line inputs (d1-d12) to set the maximum runtime for each subtask.

```
./run_single_murphi.sh 5 d1 d2 d3 d4 d5 d6
 ↪  d7 d8 d9 d10 d11 d12
```

For Experiment (6):

```
./run_single_murphi.sh 6
```

Example usage: Run the first sub-task in Experiment (4) for a maximum of 12 hours and the others for a maximum of 36 hours:

```
./run_single_murphi.sh 4 0.5 1.5 1.5 1.5
 ↪  1.5 1.5
```

Runtime expectations:. For each protocol in Experiments (2) and (6) of Table I, since they deadlock, they take relatively little time to run (about 20 to 40 minutes). Each protocol in Experiments (4) and (5) of Table I consists of several verification sub-tasks. Each sub-task takes about 72 hours to complete or run to the bound level of 48. Thus Experiment (4), the CHI protocol that has 6 sub-tasks, takes about 72x6 hours to complete. Experiment (5), the MSI and MESI protocols that have 6 sub-tasks each, takes about 72x6x2 hours to complete. In the run-all option, we set a default run time of 72 hours for each sub-task. Since model-checking takes a long time to complete, we provide the run-single option to let the user define the maximum runtime for each sub-task.

H. Expected Verification Results

For each of the model-checking experiments, Murphi will generate a result text file. There are three types of results:

1. The protocol deadlocks;

2. The model checker reaches a certain bound level and the protocol has not deadlocked;

3. The model checker completes and the protocol has no deadlocks.

The protocols in Experiments (2) and (6) should have deadlocks (result 1). The protocols in Experiments (4) and (5) should either complete or reach the bound level of 48 without error/deadlock (result 2 or 3).

We also provide a script to extract results from Murphi-generated result text files.

For all experiments:

```
python3 result_extract_murphi.py all
```

For a single experiments (2), (4), (5) or (6):

```
python3 result_extract_murphi.py 2
python3 result_extract_murphi.py 4
python3 result_extract_murphi.py 5
python3 result_extract_murphi.py 6
```

This script will judge if the results match our expectation (i.e., if they reproduce Table I) and write it to murphi_result.csv.