A Consistency Architecture for Hierarchical Shared Caches

Edya Ladan-Mozes and Charles E. Leiserson
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
{edya,cel}@mit.edu

ABSTRACT

Hierarchical Cache Consistency (HCC) is a scalable cache-consistency architecture for chip multiprocessors in which caches are shared hierarchically. HCC's cache-consistency protocol is embedded in the message-routing network that interconnects the caches, providing a distributed and scalable alternative to bus-based and directory-based consistency mechanisms. The HCC consistency protocol is "progressive" in that every message makes monotonic progress without timeouts, retries, negative acknowledgments, or retreating in any way. The latency is at most proportional to the diameter of the network. For HCC with a binary fat-tree network, the protocol requires at most 13 bits of additional state per cache line, no matter how large the system. We prove that the HCC protocol is deadlock free and provides sequential consistency.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Memory Structures—*Cache Memories, Shared Memory*; C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*

General Terms

Design, Performance, Theory

Keywords

Cache consistency, Deadlock, Fat-tree, Mapping collision, Memory hierarchy, Message race, Progressive protocol, Sequential consistency, Shared caches.

1. INTRODUCTION

Multicore technology will soon allow hundreds of processor cores to be placed on a single semiconductor die. The trend toward steeper memory hierarchies is continuing as well, leading to on-chip shared caches as a mean of exploiting instruction and data locality among the cores. Today, most chip multiprocessors (CMP's) share only their last level of cache, however. Cache consistency is usually maintained using either snoopy-bus or directory-based protocols, which typically employ timeouts, retries, or negative acknowledgments in order to avoid anomalies such as deadlock and livelock ([36] is a notable exception), and thus they do not guarantee absolute forward progress. Extending these protocols to more levels of hierarchy can be a significant design challenge.

SPAA'08, June 14–16, 2008, Munich, Germany.

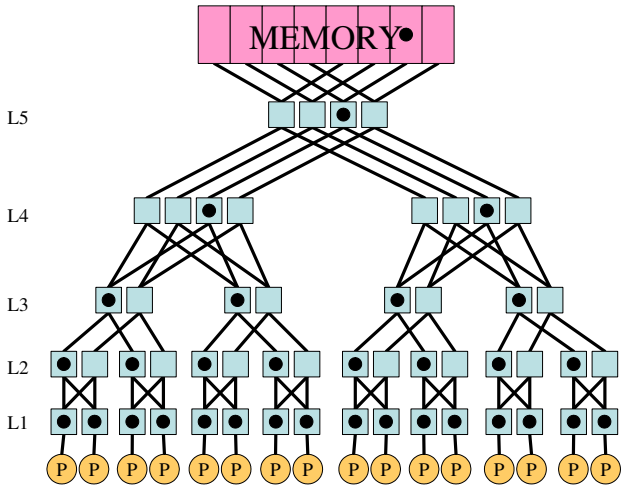

Figure 1: The HCC architecture imposed on a fat-tree interconnection network. Each memory bank is the root of a tree of caches, any of which may hold a location from the bank. The caches forming the tree rooted at one such memory bank are marked with dots.

This paper proposes a simple architectural strategy, called ***HCC***, for ***Hierarchical Cache Consistency***, which works with a variety of hierarchical networks. We give a fully distributed cache-consistency protocol which is ***progressive***: each message makes monotonic progress without timeouts, retries, or negative acknowledgments. Thus, although messages may be delayed by congestion, once sent, they never need to be resent, thereby minimizing interconnect bandwidth.

The HCC architecture consists of multiple levels of shared caches connecting the cores (or processors) to the main memory. Figure 1 shows one possible design of a butterfly fat-tree [25] in which processors connect at the leaves, banks of main memory connect to the roots, and shared caches occupy the internal nodes. The key topological property of the HCC architecture is that there exists a unique path through the network from each core to each memory bank. This ***unique-path property*** ensures that the paths from the cores to a given memory bank form a tree rooted at the memory bank. It also guarantees that for any cache and for any memory location, there is exactly one parent at the next higher level that can store the same location. In Figure 1, the caches forming such a tree for one memory bank are shown cross-hatched. Each cache is shared by its children caches in the next lower level, and can communicate only with its immediate children and parents. HCC can operate with more children and/or parents than shown in Figure 1 to meet engineering constraints, and it also supports unbalanced structures, multibutterfly-like [24] randomized connections, and a variety of optimizations, as long as the unique-path property is kept.

HCC's progressive protocol is fully distributed among the shared

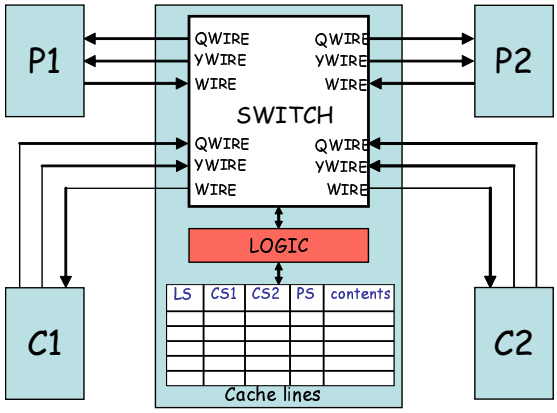

Figure 2: A shared cache node with two parents and two children.

caches. We prove that the HCC protocol supports sequential consistency[1] [23] and is deadlock free. The latency imposed by the HCC protocol is at most proportional to the diameter of the network. The HCC protocol distributes directory information among the shared caches, keeping bounded state per cache line. Each shared cache operates deterministically using only the local information it possesses. The protocol is highly asynchronous, and the system can handle any combination of messages correctly without deadlock, including message races that result from conflicting requests. The HCC protocol presented here implements the Modify/Shared/Invalid (MSI) protocol [5, 16, 33], but other protocols, such as MESI [18] and MOESI [4], can be implemented similarly.

Figure 2 depicts the general form of a cache with two parents and two children. Two incoming wires and one outgoing wire run between a cache and each of its children, and two outgoing wires and one incoming wire run from a cache to each of its parents. The cache itself contains three components: a set of ***cache lines***, a ***switch***, and ***control logic***. The cache lines maintain state information and data for the memory location stored in the cache line. The switch routes messages in and out of the cache. The control logic updates the state of the appropriate cache line and creates new messages to send in response to an incoming message dealing with a particular cache line. Our correctness and latency bounds require no assumptions about whether the cache lines are organized as fully associative, multiway associative, or direct mapped, although these decisions may affect aggregate performance and should be made judiciously depending on the engineering context [34].

Since this paper focuses on correctness and liveness issues which are generally orthogonal to the topological concerns, we present HCC assuming a binary fat-tree architecture where each cache has two children and one parent. For this exemplary unoptimized design, the HCC protocol requires only 13 additional bits of state per cache line beyond that used in an ordinary unshared cache.

Overall system operation

Communication is initiated by the processors, each of which executes an instruction stream containing loads and stores. The L1-cache attached to a processor translates loads and stores into the HCC protocol requests. Following the MSI protocol, a cache line can be written (stored) only if the state of the line in the L1-cache indicates a ***Modify*** access permission, which is the highest permission possible. A cache line can be read (load) in either a Modify or a ***Shared*** access permission, and if the state indicates that the cache line is ***Invalid***, the lowest permission, it cannot be read or written. Request messages sent from a child to its parent ask to upgrade the child's current state of the memory location. Requests from the parent to a child ask to downgrade.

The progressive HCC protocol avoids deadlock in part by properly routing messages on the three wires between caches depicted in Figure 2, depending on the type of communication. The unique-path property ensures that for a given memory location z capable of being stored in a cache i, only one of i's parents, denoted by $\text{PARENT}(i, z)$, is also capable of holding z. Request messages from a child i to its parent j are routed on $\text{QWIRE}\langle i, j\rangle$. Likewise, reply messages from a child i to its parent j are routed on $\text{YWIRE}\langle i, j\rangle$. Both requests and replies from a parent j to a child i are routed on $\text{WIRE}\langle j, i\rangle$. Each wire may contain buffering to store one or more messages, but no buffering is required. Flow-control logic stalls the sender if the wire is busy with a previous message.

A cache in the HCC architecture operates as follows. Among all the messages arriving on its incoming wires, the switch chooses a message to process, prioritizing parent messages over child messages and within child messages, replies over requests. The logic takes the message and consults the state of the appropriate cache line. If it can process the message (parent messages and child replies can always be processed), it updates the line state, and possibly provides the switch with new messages to be sent. If it cannot process the message (which can only occur for child requests), it leaves the message on the incoming wire, possibly delaying other messages that wish to use the wire. This routine, which we assume is executed atomically, then repeats. The operation of a cache can be optimized — for example, by parallelizing the logic to handle multiple messages simultaneously — but we have opted for simplicity to explain the key concepts and establish the basic correctness and liveness properties of the progressive protocol.

The remainder of this paper is organized as follows. Section 2 describes the states and messages used in HCC. Section 3 shows how states are updated, and Section 4 describes the control flow of messages. Section 5 provides graph definitions used in Section 6 to prove that HCC is deadlock free and in Section 7 to prove that HCC supports sequential consistency. Section 8 briefly reviews related work, and Section 9 offers some concluding remarks.

2. HCC STATES AND MESSAGES

This section describes the state information and messages attributes and syntax used by the HCC protocol. The state information per cache line amounts to 13 bits beyond the data itself for a binary fat-tree network.

Cache lines

The ***location*** (or ***tag***) z of a cache line is the memory location that is mapped to the cache line. For a given cache i and location z, the function $\text{CACHE-LINE}(z, i)$ maps z to a line in i. Memory locations may be mapped to different cache lines in different caches. In addition to storing the contents of the memory location, each cache line maintains the following auxiliary information:

- $LS \in \{\text{M}, \text{S}, \text{I}\}$ — The ***location state*** of a location z indicates its access permission: ***Modify*** (M), ***Shared*** (S), or ***Invalid*** (I).
- CS — The ***child state***, which is stored separately for each of the cache's children, represents the knowledge this cache has about the state of z for the particular child. In addition to the three MSI states, CS can be in the following intermediate states:
  - A ***pending*** state P indicating that a request regarding location z has been made to this child, and this cache is pending on a reply from this child.
  - A ***waiting*** state W indicating that this child is awaiting a

[1]Weaker memory models can be supported by HCC with less effort, but sequential consistency is easier for the programmer [17], and it demonstrates the efficacy of the HCC architectural strategy.

$z \backslash z'$	I	W_M	W_S	P
I	I	IW_M	IW_S	IP
S	S			SP
M	M			MP
P	P	PW_M	PW_S	
W	W			
C	C			

Figure 3: The possible values of a child state (CS).

$z \backslash z'$	N	W	P
N	N	NW	NP
P	P	PW	
W	W		WP
C	C		

Figure 4: The possible values of a parent state (PS).

reply regarding location z from this cache.

- A ***message-race*** state C indicating that this child was waiting for a reply regarding z from this cache, but in the meantime this cache received a request also regarding z from its parent.

- PS — The ***parent state*** keeps track of requests sent to the parent. The parent state can be P for pending, W for waiting, C for message race, or N for empty.

We say that the state of a line is ***steady*** if LS and CS are M, S, or I and $PS = \text{N}$. When $LS = \text{I}$ and all children states are I, we say that the state is ***all-invalid***. Each L1-cache keeps only location and parent states for each line. Each memory bank keeps only location and children states, as well as one additional bit $RS = \{M, S\}$, called the ***requested state***, which indicates the access permission requested in the message currently being processed.

Special states for mapping collisions

A ***mapping collision*** occurs when a cache receives a request from a child regarding a memory location z' that maps to the same line as a location z that is already in its cache. When two locations z and z' are mapped to the same cache line, the child state must also represent the knowledge this cache has about the state of z' for the particular child. We refer to the location that is currently stored in the cache line as z, and the location that is mapped to the same location but is not stored in cache as z'.

The states of z' can be one of the following:

- A ***waiting-for-shared*** state W_S, which indicates that this child is awaiting Shared access permission to a location z' that is mapped to the same line as location z already in the cache.

- A ***waiting-for-modify*** state W_M, which indicates that this child is awaiting Modify access permission to a location z' that is mapped to the same line as location z already in the cache.

- A ***different-location-pending*** state P, which indicates that this child must send an invalidation reply regarding location z, where z is the location currently saved in the cache and is mapped to the same line as z' (which is no longer stored in the cache and is being invalidated).

The state representation of location z' that maps to the same line as z is combined with the state of z stored in the cache. Figure 3 shows these states for CS and the combinations they may create with the state of z, and Figure 4 shows these states and combinations for PS. In both CS and PS, the first letter indicates the state regarding location z that is stored in the cache line, and the second indicates the state of another location z' that is not in the cache line. For example, in CS, MP means that the child has Modify access permission to location z, and this cache is also pending on a reply from this child regarding location z'. If z' is invalid, then only one letter is used to express the state of z. For the parent state, these additional states indicate whether the parent is awaiting a reply or if this cache is pending on a message from the parent regarding another location that maps to the same line. If z' is N, then only one letter is used to express the state of z stored in the cache line.

The overhead incurred by state information

The cache-line overhead imposed by the additional information kept by HCC in case where each cache has two children (by the unique path property each memory location stored in the cache line has only one parent) is 13 bits for a cache, calculated as follows:

LS	$CS0$	$CS1$	PS	Total
2 bits	4 bits	4 bits	3 bits	13 bits

Only 5 bits are required for each L1-cache line, since they do not have children, and only 11 bits are required for each location in memory, since they do not have parents (but do have the RS bit). These bounds can be improved by suitable encoding and by optimizing to avoid unused combinations.

Messages

A cache sends messages to and receives messages from its children and parents. A message m includes the following attributes:

- $direction \in \{C, P\}$ — The ***direction*** tells whether m was received from a child (C) or a parent (P).
- $type \in \{Q, Y\}$ — The ***message type*** indicates whether m is a request (Q) or a reply (Y).
- $location \in \mathbb{Z}$ — The ***memory location*** that the message deals with. This location may be mapped to a different cache line in each cache and is usually denoted as z.
- $access \in \{\text{M}, \text{S}, \text{I}\}$ — In a request message, the ***access permission*** indicates the access permission desired for the memory location (as in the definition of LS). In reply messages, it indicates what permission for the location is granted. If a cache line holds a location in a given state, a request to change the access permission to a lower state is a ***downgrade***. Otherwise, the request is an ***upgrade***.
- $contents \in \mathbb{Z}$ — The ***contents*** of memory location z, which is usually denoted by d, is the data stored at z. This information is only attached to reply messages.

For notational convenience, we adopt a simple syntax to describe messages. The notation $CQ_{\text{M}}\langle z\rangle$ means that the message is a request (Q) from a child (C) to obtain the contents of location z with M access permission. The message $CY_{\text{I}}\langle z, d\rangle$ means that the message is a reply from a child in which the access permission of z is downgraded to I and the contents of location z is d.

3. THE CONTROL LOGIC

The progressive HCC protocol employs a transition function to change the line states in caches and to create new messages to be sent. The transition function maintains several invariants to implement sequential consistency using an MSI strategy [33]. It copes with the hierarchical and asynchronous nature of the HCC architecture, races between messages, and eviction. The transition function exploits parallelism in the architecture, ensuring that the latency to fulfill a request is at most proportional to the network diameter.

Invariants

The HCC protocol maintains the following key invariants:

Inclusion property [7, 22]: A cache contains all the lines stored by its children. In fact, one can maintain a "weak" inclusion property where a cache contains only all the state information (but not necessarily the contents) of all the lines stored by its children. For simplicity, however, this paper assumes "strong" inclusion.

Permission-inclusion property: If a cache i holds a location z, then its parent $j = \text{PARENT}(i, z)$ must hold z with access permission at least as strong as i's. For example, j cannot evict z without first causing i to evict z (and recursively). As another example, if i

and j both hold z with access permission S and i wishes to upgrade to M, then j must also first upgrade to M.

Single-request property: If a cache i sends a request to either child or parent cache j involving a memory location z, it cannot send a second request to j involving z until after it has received a reply to its first request.

Location-mapping property: Memory locations may be mapped to different cache lines in different caches. If locations z and z' are mapped to the same cache line l_i in cache i, however, then they are also mapped to the same cache line l_j in j, where j is a child of i.

MSI property: If an L1-cache holds a location z in the M state, no other L1-cache holds z in either the M or S states.

Transition tables

The transitions describe the actions to be taken when a message is processed. Figures 5–8 describe in detail the transitions of the HCC protocol for shared caches. Since messages received from Child 0 and from Child 1 are symmetric, we omit table entries involving messages from Child 1. The state of a cache line is $(LS, CS0, CS1, PS)$. The protocol for L1-caches differs slightly from that for a regular cache, because L1-caches do not have children. Thus, the state of a line in an L1-cache is (LS, PS), and no CS fields are needed. Likewise, the protocol for the memory also differs, because the memory does not have a parent, only children. The state of a line in memory is $(LS, CS0, CS1)$ and the RS bit kept for each line in the memory.

To illustrate how the transition tables maintain the invariants, we shall walk through a simple example from a single cache's point of view. Although the progressive protocol typically involves more than one cache, we can understand how the invariants are maintained by focusing on the role of just a single cache. Section 4 takes a global view of the control flow among caches.

Suppose that cache i receives a request $CQ_{\mathrm{M}}\langle z\rangle$ from Child 0 for a location z that it does not hold but for which it has space (no eviction is necessary). The state of line $l = \text{CACHE-LINE}(z, i)$ to where z will be mapped is all-invalid: (I, I, I, N). When the request $CQ_{\mathrm{M}}\langle z\rangle$ from Child 0 is processed, transition 4 in Figure 5 changes the state to (I, W, I, P) and, to maintain the invariants, forwards the message $CQ_{\mathrm{M}}\langle z\rangle$ to i's parent. This new state of the line indicates that cache i is pending on a reply from its parent and that Child 0 is waiting for cache i to reply.

The cache continues to process messages dealing with other locations until the reply $PY_{\mathrm{M}}\langle x, d\rangle$ arrives from its parent. At this point, transition 76 in Figure 8 saves the data d in the cache, updates the state of l to (M, M, I, N) (a steady state), and forwards the message $PY_{\mathrm{M}}\langle x, d\rangle$ to Child 0. As long as the location state remains $LS = M$, cache i can manage the line in its subtree as if it were the main memory because of the permission-inclusion and MSI properties. For example, it need not communicate with its parent to invalidate the location in one subtree and give Modify permission to an L1-cache in the other subtree.

The transition tables handle more complicated situations than this simple example illustrates, but the ideas are similar. This paper includes only a partial set of the transitions for shared caches due to space limitations. The rest of this section describes how the HCC progressive protocol handles message races and mapping collisions asynchronously and in latency proportional to network diameter.

Handling message races

One particular situation, called a ***message race***, appears to threaten correctness, however. Suppose that a $CQ_{\mathrm{M}}\langle z\rangle$ request from Child 0 arrives at cache i, and the state of line $l = \text{CACHE-LINE}(z, i)$ in i is (S, S, I, N). Cache i processes this request according to transition 7, updates the state of l, and forwards $CQ_{\mathrm{M}}\langle z\rangle$ to its parent

#	Message	State	Action	New State
1	$CQ_{\mathrm{S}}\langle z\rangle$	I, I, I, N	$qp \leftarrow CQ_{\mathrm{S}}\langle z\rangle$	I, W, I, P
2	$CQ_{\mathrm{S}}\langle z\rangle$	M, I, M, N	$qc2 \leftarrow PQ_{\mathrm{S}}\langle z\rangle$	M, W, P, N
3	$CQ_{\mathrm{S}}\langle z\rangle$	S, I, S, N	$yc1 \leftarrow PY_{\mathrm{S}}\langle z, d\rangle$	S, S, S, N
4	$CQ_{\mathrm{M}}\langle z\rangle$	I, I, I, N	$qp \leftarrow CQ_{\mathrm{M}}\langle z\rangle$	I, W, I, P
5	$CQ_{\mathrm{M}}\langle z\rangle$	M, I, M, N	$qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$	M, W, P, N
6	$CQ_{\mathrm{M}}\langle z\rangle$	S, I, S, N	$qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$ $qp \leftarrow CQ_{\mathrm{M}}\langle z\rangle$	S, W, P, P
7	$CQ_{\mathrm{M}}\langle z\rangle$	S, S, I, N	$qp \leftarrow CQ_{\mathrm{M}}\langle z\rangle$	S, W, I, P
8	$CQ_{\mathrm{M}}\langle z\rangle$	S, S, S, N	$qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$ $qp \leftarrow CQ_{\mathrm{M}}\langle z\rangle$	S, W, P, P
9^{5}	$CQ_{\mathrm{M}}\langle z\rangle$	M, S, S, N	$qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$	M, W, P, N
10^{1}	$CQ_{\mathrm{S}}\langle z'\rangle$	M, I, M, N	$qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$	M, $\mathrm{IW_S}$, P, N
$11^{1,2}$	$CQ_{\mathrm{S}}\langle z'\rangle$	S, I, S, N	$qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$ $qp \leftarrow CQ_{\mathrm{S}}\langle z'\rangle$	S, $\mathrm{IW_S}$, P, NP
$12^{1,3}$	$CQ_{\mathrm{S}}\langle z'\rangle$	S, S, I, N	$qp \leftarrow CQ_{\mathrm{S}}\langle z'\rangle$	S, $\mathrm{PW_S}$, I, NP
$13^{1,3}$	$CQ_{\mathrm{S}}\langle z'\rangle$	S, S, S, N	$qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$ $qp \leftarrow CQ_{\mathrm{S}}\langle z'\rangle$	S, $\mathrm{PW_S}$, P, NP
$14^{1,3,4}$	$CQ_{\mathrm{S}}\langle z'\rangle$	M, S, S, N	$yp \leftarrow CY_{\mathrm{S}}\langle z, d\rangle$ $qp \leftarrow CQ_{\mathrm{S}}\langle z'\rangle$ $qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$	S, $\mathrm{PW_S}$, P, NP
15^{1}	$CQ_{\mathrm{M}}\langle z'\rangle$	M, I, M, N	$qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$	M, $\mathrm{IW_M}$, P, N
$16^{1,2}$	$CQ_{\mathrm{M}}\langle z'\rangle$	S, I, S, N	$qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$ $qp \leftarrow CQ_{\mathrm{M}}\langle z'\rangle$	S, $\mathrm{IW_M}$, P, NP
17^{1}	$CQ_{\mathrm{M}}\langle z'\rangle$	S, S, I, N	$qp \leftarrow CQ_{\mathrm{S}}\langle z'\rangle$	S, $\mathrm{PW_M}$, I, NP
$18^{1,3}$	$CQ_{\mathrm{M}}\langle z'\rangle$	S, S, S, N	$qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$ $qp \leftarrow CQ_{\mathrm{S}}\langle z'\rangle$	S, $\mathrm{PW_M}$, P, NP
$19^{1,3,4}$	$CQ_{\mathrm{M}}\langle z'\rangle$	M, S, S, N	$yp \leftarrow CY_{\mathrm{S}}\langle z, d\rangle$ $qp \leftarrow CQ_{\mathrm{S}}\langle z'\rangle$ $qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$	S, $\mathrm{PW_M}$, P, NP

Figure 5: Processing requests from Child 0. *Notes:* (1) Handling a request for location z' which maps to the same line as location z that is already in cache. The content of the cache line reflects location z. (2) There is no need to wait for the invalidation reply from Child 1 before requesting z' from the parent. (3) The new state indicates that this cache is pending on invalidation replies from Child 0 and Child 1 regarding location z and on its parent reply regarding z'. Child 0 is waiting for a reply from this cache regarding location z'. (4) This cache has the most updated value of z, which it sends to its parent. It cannot invalidate z since the children has it in shared state. Similar actions should be taken when the state is MIIN, MISN, and MSIN. There is no need to send an invalidation request to the child whose state is I, and thus no need to change the state to be pending on a reply for it.

$j = \text{PARENT}(i, z)$. At this point, the state of l at cache i is no longer steady. Now, suppose that before j can observe the request from i, it sends i a $PQ_{\mathrm{I}}\langle z\rangle$ request. Here, the potential for deadlock arises: before processing other messages regarding z, the parent j must wait for a reply from i to its $PQ_{\mathrm{I}}\langle z\rangle$ request, and meanwhile, before i can process another message involving z, it must wait for a reply from j to its request $CQ_{\mathrm{M}}\langle z\rangle$. In order to solve such situations, HCC maintains the following ***priority rules***:

- Parent messages can always be processed, even if the state of the line is not steady.
- Child replies can always be processed, even if the state of the line is not steady.
- Child requests can only be processed if the line's state is steady.

Since child replies are sent on different wires than child requests, they are never blocked by requests that cannot be processed.

Resuming our example, cache i processes the $PQ_{\mathrm{I}}\langle z\rangle$ request from j by following transition 52, forwarding the $PQ_{\mathrm{I}}\langle z\rangle$ request to Child 0, and changing the line state to indicate that a race has occurred. The parent j, however, cannot process the child $CQ_{\mathrm{M}}\langle z\rangle$ request from i, since the state of the line in j is not steady and the child request has lower priority (although i need not send it again).

#	Message	State	Action	New State
20	$CY_{\mathrm{I}}\langle z, d\rangle$	M, P, I, W	save d in cache $yp \leftarrow CY_{\mathrm{I}}\langle z, d\rangle$	I, I, I, N
21	$CY_{\mathrm{S}}\langle z, d\rangle$	M, P, I, W	save d in cache $yp \leftarrow CY_{\mathrm{S}}\langle z, d\rangle$	S, S, I, N
22[1]	$CY_{\mathrm{I}}\langle z, d\rangle$	M, P, W, N	save d in cache $yc2 \leftarrow PY_{\mathrm{M}}\langle z, d\rangle$	M, I, M, N
23	$CY_{\mathrm{S}}\langle z, d\rangle$	M, P, W, N	save d in cache $yc2 \leftarrow PY_{\mathrm{S}}\langle z, d\rangle$	M, S, S, N
24[2]	$CY_{\mathrm{I}}\langle z, d\rangle$	M, P, P, W	none	M, I, P, W
25[2]	$CY_{\mathrm{I}}\langle z, d\rangle$	S, P, P, W	none	S, I, P, W
26	$CY_{\mathrm{I}}\langle z, d\rangle$	S, P, I, W	$yp \leftarrow CY_{\mathrm{I}}\langle z, d\rangle$	I, I, I, N
27	$CY_{\mathrm{I}}\langle z, d\rangle$	M, P, W, N	save d in cache $yc2 \leftarrow PY_{\mathrm{M}}\langle z, d\rangle$	M, I, M, N
28[3]	$CY_{\mathrm{I}}\langle z, d\rangle$	S, P, W, P	none	S, I, W, P
29	$CY_{\mathrm{I}}\langle z, d\rangle$	S, C, I, C	$yp \leftarrow CY_{\mathrm{I}}\langle z, d\rangle$	I, W, I, P
30[2]	$CY_{\mathrm{I}}\langle z, d\rangle$	S, C, P, C	none	S, W, P, C
31[2]	$CY_{\mathrm{I}}\langle z, d\rangle$	S, P, C, C	none	S, I, C, C
32[4]	$CY_{\mathrm{I}}\langle z, d\rangle$	S, P, W, C	$yp \leftarrow CY_{\mathrm{I}}\langle z, d\rangle$	I, I, W, P
33[4]	$CY_{\mathrm{I}}\langle z, d\rangle$	M, P, C, C	$yp \leftarrow CY_{\mathrm{I}}\langle z, d\rangle$	I, I, W, P
34[5]	$CY_{\mathrm{I}}\langle z, d\rangle$	M, M, I, N	save d in cache	M, I, I, N
35[5]	$CY_{\mathrm{I}}\langle z, d\rangle$	M, S, I, N	none	M, I, I, N
36	$CY_{\mathrm{I}}\langle z, d\rangle$	M, S, S, N	none	M, I, S, N
37	$CY_{\mathrm{I}}\langle z, d\rangle$	S, S, S, N	none	S, I, S, N
38[5]	$CY_{\mathrm{I}}\langle z, d\rangle$	S, S, I, N	none	S, I, I, N
39[1,5]	$CY_{\mathrm{I}}\langle z, d\rangle$	M, P, I, N	save d in cache	M, I, I, N
40[5]	$CY_{\mathrm{I}}\langle z, d\rangle$	S, P, I, N	none	S, I, I, N
41	$CY_{\mathrm{I}}\langle z, d\rangle$	S, P, P, N	none	S, I, P, N

Figure 6: Processing replies from Child 0 when the state involves only one memory location. When a reply arrives from a child with new data for z and a reply is sent to the parent, the data in the reply to the parent is that received in the reply from the child. *Notes:* (1) Child 0 had the line in M state. (2) Pending on Child 1. (3) Pending on the parent. (4) Child 1 already replied to the conflicting message. (5) The cache is free to evict the line now.

When a reply from Child 0 arrives at i, transition 29 is triggered, the reply is forwarded to j, and i's state is reconstructed as if the race had not occurred. These actions leave i waiting for the reply from j to its original request. Meanwhile, j can now process the reply from i to its original request, because the reply from i is sent on the reply wire $\textsc{ywire}\langle i, j\rangle$ and thus is not blocked by the original request from i, which was sent on the request wire $\textsc{qwire}\langle i, j\rangle$.

It is possible that when cache i processes the $CQ_{\mathrm{M}}\langle z\rangle$ request, the state of l is as depicted in transition 8. Then, in addition to the $CQ_{\mathrm{M}}\langle z\rangle$ request sent to j, cache i sends a $PQ_{\mathrm{I}}\langle z\rangle$ request to Child 1. If a race now occurs, i sends a $PQ_{\mathrm{I}}\langle z\rangle$ request only to Child 0 (transition 53), thereby preserving the single-request property, since i has already sent a request to Child 1 involving the same location and cannot send another one. After cache i receives replies from both children, it sends a $CY_{\mathrm{I}}\langle z, d\rangle$ reply to its parent j and then waits for a reply from j to its original $CQ_{\mathrm{M}}\langle z\rangle$ request. Because the memory banks have no parents, their transitions need not deal with races.

Handling mapping collisions

The HCC protocol handles mapping collisions differently when location z (which is already in the cache) is in Shared access permission or in Modify access permission. The goal in both cases is to allow HCC to process each request in latency that is at most proportional to the diameter of the network. The state of the cache line can represent the state of at most two memory locations z and z'. Any other request from a child regarding another location z'' that is mapped to the same line as z and z' waits until the state of the line is steady (and thus represents the state of one memory location). Also, the parent may send requests to its child only if it is indicated in the state of the line at the parent that this child has the line in Shared or Modify access permission. Therefore, the parent never sends a request regarding z'' that may be mapped to the same location as z and z', because the child cannot have z'' in its cache.

#	Message	State	Action	New State
42	$PQ_{\mathrm{I}}\langle z\rangle$	M, I, M, N	$qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$	M, I, P, W
43	$PQ_{\mathrm{I}}\langle z\rangle$	M, M, I, N	$qc1 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$	M, I, P, W
44	$PQ_{\mathrm{I}}\langle z\rangle$	M, S, S, N	$qc1 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$ $qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$	M, P, P, W
45	$PQ_{\mathrm{S}}\langle z\rangle$	M, I, M, N	$qc2 \leftarrow PQ_{\mathrm{S}}\langle z\rangle$	M, I, P, W
46	$PQ_{\mathrm{S}}\langle z\rangle$	M, M, I, N	$qc1 \leftarrow PQ_{\mathrm{S}}\langle z\rangle$	M, P, I, W
47	$PQ_{\mathrm{S}}\langle z\rangle$	M, S, S, N	$yp \leftarrow CY_{\mathrm{S}}\langle z, d\rangle$	S, S, S, N
48	$PQ_{\mathrm{I}}\langle z\rangle$	S, I, S, N	$qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$	S, I, P, W
49	$PQ_{\mathrm{I}}\langle z\rangle$	S, S, I, N	$qc1 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$	S, P, I, W
50	$PQ_{\mathrm{I}}\langle z\rangle$	S, S, S, N	$qc1 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$ $qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$	S, P, P, W
51[1]	$PQ_{\mathrm{I}}\langle z\rangle$	I, I, I, N	none	I, I, I, N
52	$PQ_{\mathrm{I}}\langle z\rangle$	S, W, I, P	$qc1 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$	S, C, I, C
53	$PQ_{\mathrm{I}}\langle z\rangle$	S, W, P, P	$qc1 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$	S, C, P, C
54	$PQ_{\mathrm{I}}\langle z\rangle$	S, I, W, P	$qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$	S, I, C, C
55	$PQ_{\mathrm{I}}\langle z\rangle$	S, P, W, P	$qc2 \leftarrow PQ_{\mathrm{I}}\langle z\rangle$	S, P, C, C
56[2]	$PQ_{\mathrm{I}}\langle z\rangle$	M, W, P, N	$qp \leftarrow CQ_{\mathrm{M}}\langle z\rangle$	M, C, P, C
57	$PQ_{\mathrm{I}}\langle z\rangle$	M, P, I, N	none	M, P, I, W
58	$PQ_{\mathrm{I}}\langle z\rangle$	M, I, P, N	none	M, I, P, W
59	$PQ_{\mathrm{I}}\langle z\rangle$	S, P, I, N	none	S, P, I, W
60	$PQ_{\mathrm{I}}\langle z\rangle$	S, I, P, N	none	S, I, P, W
61	$PQ_{\mathrm{I}}\langle z\rangle$	S, P, P, N	none	S, P, P, W
62[3,5]	$PQ_{\mathrm{I}}\langle z\rangle$	M, $\mathrm{IW_S}$,P, N	none	M, $\mathrm{IW_S}$, P, W
63[3,5]	$PQ_{\mathrm{I}}\langle z\rangle$	S, $\mathrm{IW_S}$, P, NP	none	S, $\mathrm{IW_S}$, P, PW
64[3,5]	$PQ_{\mathrm{I}}\langle z\rangle$	S, $\mathrm{PW_S}$, I, NP	none	S, $\mathrm{PW_S}$, I, PW
65[3,5]	$PQ_{\mathrm{I}}\langle z\rangle$	S, $\mathrm{PW_S}$, P, NP	none	S, $\mathrm{PW_S}$, P, PW
66[3,5]	$PQ_{\mathrm{I}}\langle z\rangle$	M, $\mathrm{IW_M}$, P, N	none	M, $\mathrm{IW_M}$, P, W
67[3,5]	$PQ_{\mathrm{I}}\langle z\rangle$	S, $\mathrm{IW_M}$, P, NP	none	S, $\mathrm{IW_M}$, P, PW
68[3,5]	$PQ_{\mathrm{I}}\langle z\rangle$	S, $\mathrm{PW_M}$, I, NP	none	S, $\mathrm{PW_M}$, I, PW
69[3,5]	$PQ_{\mathrm{I}}\langle z\rangle$	S, $\mathrm{PW_M}$, P, NP	none	S, $\mathrm{PW_M}$, P, PW
70[3,4,5]	$PQ_{\mathrm{I}}\langle z\rangle$	S, S, IP, N	none	S, S, IP, NW
71[3,4,5]	$PQ_{\mathrm{I}}\langle z\rangle$	S, I, SP, N	none	S, I, SP, NW
72[3,4,5]	$PQ_{\mathrm{I}}\langle z\rangle$	M, M, IP, N	none	M, M, IP, NW
73[3,4,5]	$PQ_{\mathrm{I}}\langle z\rangle$	M, I, MP, N	none	M, I, MP, NW

Figure 7: Processing requests from the parent. *Notes:* (1) Evicted line. (2) This cache receives a request to evict z before z is sent to Child 0 (still pending on Child 1 invalidation reply). The cache sends a request to the parent to upgrade to M state, (otherwise the cache would not be pending on Child 1) and indicates a conflict. (3) Processing an invalidation request for z while already in the process of evicting z. (4) In this case z and z' do not map to the same line at the parent cache. The content of the line in this cache corresponds to location z'. (5) Similar transitions exist when the state and new state for Child 0 and Child 1 are switched.

When the access permission of z is Shared: Suppose that Child 0 sends the request $CQ_{\mathrm{M}}\langle z'\rangle$ and z' maps to the same line as a location z already in the cache. If the state of the line indicates that Child 1 has a shared access permission to it, the cache sends an invalidation request to Child 1, but it also forwards the request from Child 0 for z'. If Child 0 also holds z in Shared access permission, then the new state of the line indicates that Child 0 still needs to invalidate location z, but no message needs to be sent. When all the invalidation replies that are needed (as indicated by the state of the line) arrive, the cache forward an invalidation reply for location z to the parent. In this way, there is no need to wait for the invalidation process of z to complete before processing the request

#	Message	State	Action	New State
74	$PY_{\mathrm{S}}\langle z, d\rangle$	I, W, I, P	save d in cache $yc1 \leftarrow PY_{\mathrm{S}}\langle z, d\rangle$	S, I, S, N
75	$PY_{\mathrm{S}}\langle z, d\rangle$	I, I, W, P	save d in cache $yc2 \leftarrow PY_{\mathrm{S}}\langle z, d\rangle$	S, I, S, N
76	$PY_{\mathrm{M}}\langle z, d\rangle$	I, W, I, P	save d in cache $yc1 \leftarrow PY_{\mathrm{M}}\langle z, d\rangle$	M, M, I, N
77	$PY_{\mathrm{M}}\langle z, d\rangle$	I, I, W, P	save d in cache $yc2 \leftarrow PY_{\mathrm{M}}\langle z, d\rangle$	M, I, M, N
78[1]	$PY_{\mathrm{M}}\langle z, d\rangle$	S, W, I, P	save d in cache $yc1 \leftarrow PY_{\mathrm{M}}\langle z, d\rangle$	M, M, I, N
79[1]	$PY_{\mathrm{M}}\langle z, d\rangle$	S, I, W, P	save d in cache $yc2 \leftarrow PY_{\mathrm{M}}\langle z, d\rangle$	M, I, M, N
80[2]	$PY_{\mathrm{M}}\langle z, d\rangle$	S, W, P, P	none	M, W, P, N
81	$PY_{\mathrm{M}}\langle z, d\rangle$	S, P, W, P	none	M, P, W, N
82[3]	$PY_{\mathrm{S}}\langle z', d\rangle$	S, $\mathrm{IW_S}$, P, NP	save d in cache $yc1 \leftarrow PY_{\mathrm{S}}\langle z', d\rangle$	S, S, IP, N
83[3]	$PY_{\mathrm{S}}\langle z', d\rangle$	S, P, $\mathrm{IW_S}$, NP	save d in cache $yc2 \leftarrow PY_{\mathrm{S}}\langle z', d\rangle$	S, IP, S, N
84[3]	$PY_{\mathrm{S}}\langle z', d\rangle$	S, $\mathrm{PW_S}$, I, NP	save d in cache $yc1 \leftarrow PY_{\mathrm{S}}\langle z', d\rangle$	S, SP, I, N
85[3]	$PY_{\mathrm{S}}\langle z', d\rangle$	S, I, $\mathrm{PW_S}$, NP	save d in cache $yc2 \leftarrow PY_{\mathrm{S}}\langle z', d\rangle$	S, I, SP, N
86[3]	$PY_{\mathrm{S}}\langle z', d\rangle$	S, $\mathrm{PW_S}$, P, NP	save d in cache $yc1 \leftarrow PY_{\mathrm{S}}\langle z', d\rangle$	S, SP, IP, N
87[3]	$PY_{\mathrm{S}}\langle z', d\rangle$	S, P, $\mathrm{PW_S}$, NP	save d in cache $yc2 \leftarrow PY_{\mathrm{S}}\langle z', d\rangle$	S, IP, SP, N
88[3]	$PY_{\mathrm{M}}\langle z', d\rangle$	S, $\mathrm{IW_M}$, P, NP	save d in cache $yc1 \leftarrow PY_{\mathrm{M}}\langle z', d\rangle$	M, M, IP, N
89[3]	$PY_{\mathrm{M}}\langle z', d\rangle$	S, P, $\mathrm{IW_M}$, NP	save d in cache $yc2 \leftarrow PY_{\mathrm{M}}\langle z', d\rangle$	M, IP M, N
90[3]	$PY_{\mathrm{M}}\langle z', d\rangle$	S, $\mathrm{PW_M}$, I, NP	save d in cache $yc1 \leftarrow PY_{\mathrm{M}}\langle z', d\rangle$	M, MP, I, N
91[3]	$PY_{\mathrm{M}}\langle z', d\rangle$	S, I, $\mathrm{PW_M}$, NP	save d in cache $yc2 \leftarrow PY_{\mathrm{M}}\langle z', d\rangle$	M, I, MP, N
92[3]	$PY_{\mathrm{M}}\langle z', d\rangle$	S, $\mathrm{PW_M}$, P, NP	save d in cache $yc1 \leftarrow PY_{\mathrm{M}}\langle z', d\rangle$	M, MP, IP, N
93[3]	$PY_{\mathrm{M}}\langle z', d\rangle$	S, P, $\mathrm{PW_M}$, NP	save d in cache $yc2 \leftarrow PY_{\mathrm{M}}\langle z', d\rangle$	M, IP MP, N

Figure 8: Processing replies from the parent. *Notes:* (1) Saving d is unnecessary, because it is already in cache. (2) Pending on Child 1 to preserve the MSI property. (3) No need to wait for the invalidation reply from Child 1 regarding location z. The new state indicates that the invalidation reply has not yet been received, however. The content of the line now reflects location z'. Child requests regarding locations mapping to the same line will wait.

for z'. Rather, they are run in parallel, thus achieving latency that is at most proportional to the network's diameter.

When the access permission of z is Modified: Suppose that Child 0 sends either a $CQ_{\mathrm{M}}\langle z'\rangle$ or a $CQ_{\mathrm{S}}\langle z'\rangle$ request and a location z' maps to the same line as z. Then, the cache first invalidates z at Child 1 and forwards the invalidation reply to the parent. After that, it sends the request for z' to the parent. Since there can only be at most one path leading to the L1-cache that has the Modified access permission for z, the delay imposed on the request for z' is at most proportional to the diameter of the network.

When a mapping collision occurs at an L1-cache, the L1-cache evicts the location currently in the cache before requesting the new location that is mapped to the same line. Also, since the memory is by definition big enough to hold all memory locations, no mapping collisions can occur in the memory.

4. CONTROL FLOW

To prove strong properties of the HCC architecture, we formalize the operation of caches beyond the simple execution of transition tables as described in Section 3. We model the operation of the switch and the priority logic by which messages are chosen for processing as a concurrent event-driven program containing a set of continuation-passing handlers. The concurrent program can be reduced to a resource-limited state machine within each cache. Section 5 uses this program to produce a computation graph for the execution of the progressive protocol, which is used in Sections 6 and 7 to prove deadlock freedom and sequential consistency.

Resources, locking, and synchronization

During its operation, a cache uses different resources. To keep the locations' states, the cache manages a set of cache lines. To send and receive messages, the cache manages several wires. Specifically, the HCC resources in each cache i are the following:

- Cache i's lines.
- QWIRE$\langle i, j\rangle$ — The wire on which cache i sends request messages to its parent cache j.
- YWIRE$\langle i, j\rangle$ — The wire on which cache i sends reply messages to its parent cache j.
- WIRE$\langle i, j\rangle$ — The wire on which cache i sends requests and replies to its child cache j.
- EWIRE$\langle i\rangle$ — A nonexistent wire attached to all L1-caches.

The message sent on a wire QWIRE$\langle i, j\rangle$ is denoted QWIRE$\langle i, j\rangle$ $.msg$. Each wire maintains extra state called ***action storage***. While processing a message, a cache may create new messages to send. These messages are buffered in the action storage associated with the input wire on which the message arrived until they can be delivered to the appropriate output wire. The action storage associated with input wires contains the following fields:

- Message yp — a reply to a parent,
- Message $yc0$ — a reply to Child 0,
- Message $yc1$ — a reply to Child 1,
- Message qp — a request to a parent,
- Message $qc0$ — a request to Child 0,
- Message $qc1$ — a request to Child 1.

To synchronize among accesses to shared resources (cache lines, wires, and their action storage), a lock is attached to each resource. Locking is governed by three procedures: ACQUIRE, RELEASE, and TRY-ACQUIRE. Whenever an ACQUIRE is called on some resource, it blocks the calling thread until the resource's lock is acquired. The lock is released by calling RELEASE on the resource. The TRY-ACQUIRE attempts to acquire the lock, but if it fails, it does not block. Instead, TRY-ACQUIRE always returns immediately with an indication whether the lock was successfully acquired.

The handlers employ a collection of service routines:

- CACHE-LINE(i, z) — The line in cache i to which memory location z is mapped.
- PARENT(i, z) — The parent of cache i with respect to memory location z.
- CHILD0(i) — The cache identified as Child 0 of cache i
- CHILD1(i) — The cache identified as Child 1 of cache i.
- IS-L1-CACHE(i) — Indicating whether cache i is an L1-cache.
- CHANGE-STATE(i, j, w) — Atomically updates the state of line $l =$ CACHE-LINE$(j, wmsg.location)$ according to the transition tables in Figures 5–8 and places the messages to send in wire w's action storage. The fields of the action storage that are not explicitly set by CHANGE-STATE are set to NIL.
- IS-STEADY-STATE(i, l) — A predicate indicating whether line l in cache i is in a steady state.

The resources used by the HCC code consist only of the cache lines, the wires that interconnect the caches, and the wires' limited

```
Handler CHILD0-REQ (Cache from, Cache to)
  requires QWIRE⟨from, to⟩
 1  m ← QWIRE⟨from, to⟩.msg
 2  z ← m.location
 3  l ← CACHE-LINE(to, z)
 4  p ← PARENT(to, z)
 5  c1 ← CHILD1(to)
 6  ACQUIRE(l)
 7  CHANGE-STATE(from, to, QWIRE⟨from, to⟩)
 8  if yc0 ≠ NIL
 9     then ACQUIRE(WIRE⟨to, from⟩)
10          WIRE⟨to, from⟩.msg ← yc0
11          RELEASE(QWIRE⟨from, to⟩)
12          RELEASE(l)
13     else if qp ≠ NIL
14             then ACQUIRE(QWIRE⟨to, p⟩)
15                  QWIRE⟨to, p⟩.msg ← qp
16          if qc1 ≠ NIL
17             then ACQUIRE(WIRE⟨to, c1⟩)
18                  WIRE⟨to, c1⟩.msg ← qc1
19          RELEASE(QWIRE⟨from, to⟩)
20  dispatch
```

Figure 9: The CHILD0-REQ handler is dispatched to execute in cache to when a request message m is sent from its Child 0. Preconditions for executing the handler include (1) the wire lock QWIRE$\langle from, to\rangle$ is held and (2) message m is enqueued on the wire.

```
Handler PARENT-REPLY (Cache from, Cache to)
  requires WIRE⟨from, to⟩
 1  m ← WIRE⟨from, to⟩.msg
 2  z ← m.location
 3  l ← CACHE-LINE(to, z)
 4  c0 ← CHILD0(to)
 5  c1 ← CHILD1(to)
 6  CHANGE-STATE(from, to, WIRE⟨from, to⟩)
 7  if yc0 ≠ NIL
 8     then ACQUIRE(WIRE⟨to, c0⟩)
 9          WIRE⟨to, c0⟩.msg ← yc0
10          RELEASE(WIRE⟨from, to⟩)
11          RELEASE(l)
12  elseif yc1 ≠ NIL
13     then ACQUIRE(WIRE⟨to, c1⟩)
14          WIRE⟨to, c1⟩.msg ← yc1
15          RELEASE(WIRE⟨from, to⟩)
16          RELEASE(l)
17     else RELEASE(WIRE⟨from, to⟩)
18          if IS-L1-CACHE(to)
19             then RELEASE(l)
20  dispatch
```

Figure 10: The PARENT-REPLY handler executes in cache to when a reply message m is sent from its parent $from$. Preconditions for executing the handler include (1) the wire lock WIRE$\langle from, to\rangle$ is held, (2) message m is enqueued on the wire, and (3) the lock on l is held.

```
Handler PARENT-REQ (Cache from, Cache to)
  requires WIRE⟨from, to⟩
 1  m ← WIRE⟨from, to⟩.msg
 2  z ← m.location
 3  l ← CACHE-LINE(to, z)
 4  c0 ← CHILD0(to)
 5  c1 ← CHILD1(to)
 6  success ← TRY-ACQUIRE(l)
 7  CHANGE-STATE(from, to, WIRE⟨from, to⟩)
 8  if yp ≠ NIL
 9     then ACQUIRE(YWIRE⟨to, from⟩)
10          YWIRE⟨to, from⟩.msg ← yp
11          RELEASE(WIRE⟨from, to⟩)
12          if success
13             then RELEASE(l)
14     else if qc0 ≠ NIL
15             then ACQUIRE(WIRE⟨to, c0⟩)
16                  WIRE⟨to, c0⟩.msg ← qc0
17          if qc1 ≠ NIL
18             then ACQUIRE(WIRE⟨to, c1⟩)
19                  WIRE⟨to, c1⟩.msg ← qc1
20          RELEASE(WIRE⟨from, to⟩)
21  dispatch
```

Figure 11: The PARENT-REQ handler executes in cache to when a request message m is sent from one of its children. Preconditions for executing the handler include (1) wire lock WIRE$\langle from, to\rangle$ is held and (2) message m is enqueued on the wire.

```
Handler CHILD0-REPLY (Cache from, Cache to)
  requires YWIRE⟨from, to⟩
 1  m ← YWIRE⟨from, to⟩.msg
 2  z ← m.location
 3  l ← CACHE-LINE(to, z)
 4  p ← PARENT(to)
 5  c1 ← CHILD1(to)
 6  success ← TRY-ACQUIRE(l)
 7  CHANGE-STATE(from, to, YWIRE⟨from, to⟩)
 8  if yp ≠ NIL
 9     then ACQUIRE(YWIRE⟨to, p⟩)
10          YWIRE⟨to, p⟩.msg ← yp
11          if qp ≠ NIL
12             then ACQUIRE(QWIRE⟨to, p⟩)
13                  QWIRE⟨to, p⟩.msg ← qp
14                  RELEASE(YWIRE⟨from, to⟩)
15             else RELEASE(YWIRE⟨from, to⟩)
16                  if IS-STEADY-STATE(to, l)
17                     then RELEASE(l)
18  elseif yc1 ≠ NIL
19     then ACQUIRE(WIRE⟨to, c1⟩)
20          WIRE⟨to, c1⟩.msg ← yc1
21          RELEASE(YWIRE⟨from, to⟩)
22          RELEASE(l)
            //all other cases fail hence success = TRUE
23     else if success
24             then RELEASE(YWIRE⟨from, to⟩)
25                  RELEASE(l)
26  dispatch
```

Figure 12: The CHILD0-REPLY handler executes in cache to when a reply message m is sent from its Child 0 $from$. Preconditions for executing the handler include (1) the wire lock YWIRE$\langle from, to\rangle$ is held and (2) message m is enqueued on the wire. Note that if $success$ is TRUE, then no messages are dispatched.

action storage. Therefore, the program can be reduced to a limited-resource state machine. We have implemented a simplified version of the HCC progressive protocol using Bluespec [6].

The HCC event-driven code

The operation of the HCC progressive protocol is organized as a set of ***handlers***, which are blocks of code that communicate using continuation passing. Although the entire HCC code is too large to present, Figures 9–13 give the pseudocode for a subset of the

```
Handler LOAD (Cache from, Location z)
 1  l ← CACHE-LINE(from, z)
 2  ACQUIRE(EWIRE⟨from⟩)
 3  ACQUIRE(l)
 4  to ← PARENT(from, z)
 5  m ← CQ_S⟨z⟩
 6  CHANGE-STATE(NIL, from, EWIRE⟨from⟩)
 7  if yp ≠ NIL
 8     then ACQUIRE(YWIRE⟨from, to⟩)
 9          YWIRE⟨from, to⟩.msg ← yp
10  if qp ≠ NIL
11     then ACQUIRE(QWIRE⟨from, to⟩)
12          QWIRE⟨from, to⟩.msg ← qp
13  RELEASE(EWIRE⟨from⟩)
14  dispatch
```

Figure 13: The LOAD procedure invoked on an L1-cache $from$ on memory location z. The STORE procedure is identical to LOAD except that Line 5 is replaced with $m \leftarrow \text{CQ}_M\langle z\rangle$.

handlers that convey the essential ideas.

A handler is invoked when a cache receives a message on an incoming wire. The specific handler invoked depends on the type of message, and it assumes that the sender properly acquired the lock on the wire before dispatching the message. The handler changes the cache-line state of the location named in the message according to the transition tables. This change is performed atomically by the hardware within the cache, including the storing of messages in the action storage. If the transition function indicates that outgoing messages need to be sent, the cache acquires locks on the outgoing wires it needs and places the messages on them. Then, it releases the lock on the incoming wire, and, if appropriate, releases the line lock as well. Finally, it dispatches the messages to their destinations, causing the appropriate handlers to be processed by the destination caches. If more than one handler is available to run in a cache, the cache picks one, prioritizing parent messages over child messages and, within child messages, replies over requests.

The code for handling parent requests in Figure 11 reflects the higher priority of parent requests using TRY-ACQUIRE on a line instead of the regular ACQUIRE. In Line 6, if the lock is already taken, then a race has occurred. The update of the state of the line

in Line 7 reflects this race. The lock on the line is released in Lines 12–13 only if the TRY-ACQUIRE in Line 6 was successful.

The TRY-ACQUIRE function is also used in Figure 12 Line 6 to handle eviction. Suppose that a child *from* needs to evict location z from the cache to accommodate another request. Then, it acquires the reply wire YWIRE$\langle from, to\rangle$ to its parent to, places the reply CY$_I\langle z, d\rangle$ on the wire, which invokes CHILD0-REPLY$(from, to)$ in cache to after messages are dispatched. Cache to does not "expect" this message, however, since it never sent a request to $from$, and therefore the lock on $l =$ CACHE-LINE(to, z) may not be held. Therefore, it first locks l before it can process the reply. If it succeeds in locking l, this reply only evicts z from the cache $from$ and (as indicated by the transition function) does not generate any messages. Hence, if the lock was successfully acquired, after the update of the state, it is released. If the lock was not successfully acquired, it indicates that some other request regarding line l was being processed. Nevertheless, since replies from the child have higher priority than requests, they can be processed even if the lock was not acquired by the handler.

5. COMPUTATION GRAPHS

Based on the control flow code in Section 4, this section provides graph definitions such as activity graphs, various predicates on their edges, and how activity graphs form a computation graph. These definitions are used in Sections 6 and 7 to prove that HCC is deadlock-free and implements sequential consistency.

Let G be a ***directed graph***, where $V(G)$ is the set of vertices belonging to G and $E(G)$ is the set of edges belonging to G. If $e = (u, v) \in E(G)$, where $u, v \in V(G)$, we call u the ***tail*** of e and v the ***head*** of e, written $u =$ tail(e) and $v =$ head(e), respectively. An edge $e \in E(G)$ ***connects to*** $e' \in E(G)$ if head$(e) =$ tail(e'). A ***path*** p of length $n \geq 0$ from a vertex $v_0 \in V(G)$ to a vertex $v_n \in V(G)$ is a sequence $\langle e_1, e_2, \ldots, e_n\rangle$ of edges in $E(G)$ such that e_1 leaves v_0, e_n enters v_n, and for $i = 1, \ldots, n-1$, we have that e_i connects to e_{i+1}. For $v, v' \in V(G)$ and $e, e' \in E(G)$, if there exists a path from v to v', we write $v \preceq_G v'$; if there exists a path from head(e) to tail(e'), we write $e \preceq_G e'$; if there exists a path from v to tail(e), we write $v \preceq_G e$; and if there exists a path from head(e) to $v' \in V(G)$, we write $e \preceq_G v'$. A directed graph is a ***dag*** if it contains no cycles. A dag is ***rooted*** if there exists a unique vertex root$(G) \in V(G)$ such that for all $e \in E(G)$, we have root$(G) \preceq_G e$. A ***cycle*** is a positive-length path from a vertex v to itself. The ***in-degree*** of a vertex $v \in V(G)$ is in-deg$(v) = |\{e \in E(G) : v =$ head$(e)\}|$. Likewise, the ***out-degree*** of a vertex $v \in V(G)$ is out-deg$(v) = |\{u \in V(G) : u =$ tail$(e)\}|$. A vertex $v \in V(G)$ is a ***fork*** if out-deg$(v) > 1$ and a ***join*** if in-deg$(v) > 1$. If out-deg$(v) = 0$, then v is a ***sink***.

We model the execution of a concurrent event-driven program as a rooted ***activity graph*** G on a set L of locks and a set Z of memory locations. Whenever a handler executes an instruction, an edge e corresponding to the instruction is added to $E(G)$ that connects from the previous instruction executed by the handler. For each edge $e \in E(G)$, we define the ***executer*** of e to be the cache that executes e's instruction. When **dispatch** is called from a handler, a parallel execution is forked for each message delivered to another cache, and a ***dispatch edge*** is created from the last instruction of the handler to the first instruction of each new handler invoked.

Within a cache, if a handler operating on a location z does not generate any messages for other caches, it is defined to be a ***joiner***. The transition function ensures that this situation only occurs when the cache line is pending on more than one reply. If a subsequent handler operates on location z and does generate a message and the state of the cache line becomes steady, we call it a ***last updater***. A

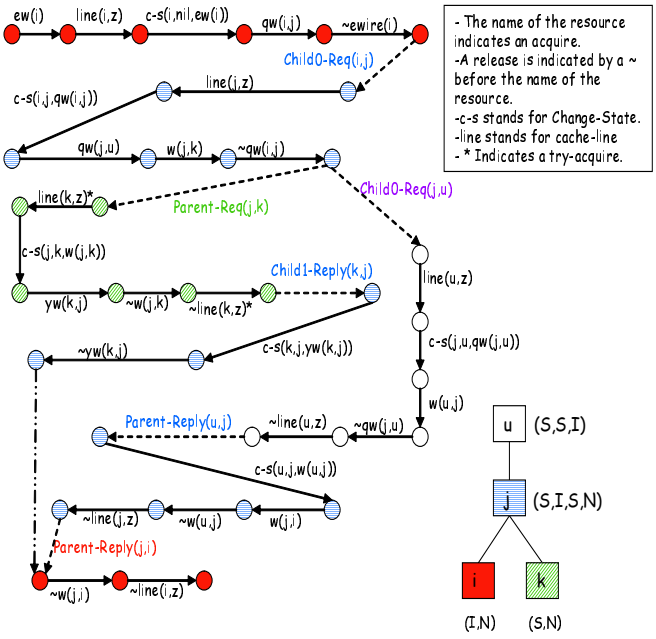

Figure 14: An example activity created by the execution of STORE(i, z). A lock acquisition is shown by naming the resource, and a lock release is shown by prepending a tilde to the resource name. When a handler is invoked, its name is also shown. The dispatch edges are dashed and the join edge is marked by a dash followed by two dots.

dispatch edge connects the last updater to the first instruction of the handler it invokes, but we also create a ***join edge*** connecting the last instruction of each joiner to the first instruction of the handler invoked by the last updater.

We define the following predicates on edges of an activity graph:

- ACQ(e, l) — e successfully executes ACQUIRE(l) or TRY-ACQUIRE(l) ;
- REL(e, l) — e executes RELEASE(l);
- W(e, z) — either e executes Line 7 of PARENT-REQ$(from, to)$, where to is an L1-cache, the message on the wire involves location z, and CACHE-LINE$(to, z).LS = M$ before the change; or e executes Line 6 of handler LOAD$(from, z')$ or STORE$(from, z')$, where CACHE-LINE$(from, z')$ is occupied with location z and CACHE-LINE$(from, z').LS = M$ before the change.
- R(e, z) — either e executes Line 7 of PARENT-REQ$(from, to)$, where to is an L1-cache, the message on the wire involves location z, and CACHE-LINE$(to, z).LS = S$ before the change; or e executes Line 6 of handler LOAD$(from, z')$ or STORE$(from, z')$, where CACHE-LINE$(from, z')$ is occupied with location z and CACHE-LINE$(from, z').LS = S$ before the change.

For an edge $e \in E(G)$ and a memory location $z \in Z$, we define RW$(e, z) =$ R$(e, z) \vee$ W(e, z), and we say that e ***accesses*** z. For any predicate $P : S \to \{$TRUE, FALSE$\}$ on a set S, we define $P(S) \subseteq S$ to be $\{s \in S : P(s)\}$. For example, W$(E(G), z)$ is the set of edges that write to z.

Figure 14 depicts an activity graph generated by invoking a handler STORE(i, z) in a given state of HCC presented in the figure. Note that STORE(i, z) is identical to LOAD(i, z) except in one line as explained in Figure 13.

For an edge $e \in E(G)$ that releases a lock $l \in L$, define the ***region ender*** of e as the vertex ender$(e) =$ head(d) if e connects to a dispatch edge d and head(e) is not a fork node, and as ender$(e) =$ head(e) otherwise. An edge $e_1 \in E(G)$ that acquires a lock $l \in L$

and an edge $e_2 \in E(G)$ that releases l form a ***critical region*** if, for all paths p from root(G) to ender(e_2), we have $e_1 \in p$ and for all edges $e \in E(G)$ such that $e \neq e_2$ and $e_1 \prec_G e \prec_G$ ender(e_2), we have $\neg$ACQ(e, l) and $\neg$REL(e, l). The critical region formed by e_1 and e_2 is $\mathrm{CR}(e_1, e_2, l) = \{e \in E(G) : e_1 \preceq_G e \preceq_G \text{ender}(e_2)\}$, and we say that an edge $e \in \mathrm{CR}(e_1, e_2, l)$ ***holds*** lock l. It follows that two critical regions whose edges hold the same lock are disjoint. Note that if a release of a lock connects to only one dispatch edge and the state of the line that was updated before the release of the lock becomes steady, then this dispatch edge was executed by the last updater (there may or may not be joiners). If the lock was acquired before a fork, then the definition of the critical region guarantees that the lock is held on the paths executed by the joiners as well, even though the release of the lock is performed only on the path of the last updater.

A ***well-structured activity*** G on a set L of locks and a set Z of memory locations is an activity in which for all $e_2 \in E(G)$ such that REL(e_2, l), there exists an edge $e_1 \in E(G)$ and lock $l \in L$ such that $\mathrm{CR}(e_1, e_2, l)$ is a critical region. An activity G on a strictly linearly ordered set $\{L, >\}$ of locks ***respects*** the ordering of the locks if for all $e_1, e', e_2 \in E(G)$ and $l, l' \in L$ such that $e' \in \mathrm{CR}(e_1, e_2, l)$ and e' executes ACQUIRE(l'), we have $l' > l$. Note that we do not consider executions of TRY-ACQUIRE(l') as violating the order of locks even if $l' < l$, because e' need never wait for the lock acquisition. A ***legal activity*** is a well-structured activity that respects the ordering of locks.

A ***start-up graph*** over a set Z of memory locations is a dag G_0 such that $V(G_0) = \{s_0, s_1\}$, $E(G_0) = \{(s_0, s_1)\}$, and for all $z \in Z$, we have $\mathrm{W}((s_0, s_1), z)$. A ***computation graph*** over a set L of locks and a set Z of memory locations is a dag $CG = G_0 \cup G_1 \cup G_2 \cup \cdots \cup G_n$, where G_0 is a start-up graph and for $i = 1, 2, \ldots, n$, each G_i is an activity on L and Z, and all activities are disjoint. A ***legal computation*** is a computation in which all activities are legal.

Each activity is initiated by a LOAD or STORE handler executed in a processor. We assume that the processors obey the ***order-consumption property***, which requires each processor to consume memory locations in the order it requests them. Multiple activities initiated by LOAD and STORE handlers can be "in flight" simultaneously, but the processor executes the instructions in the program order. The proper ordering of activities initiated by the same processor can be represented by additional edges in CG.

A t-step ***execution*** of a computation graph CG is a mapping $\delta : E(CG) \to \{1, 2\,..\,t\} \cup \{\infty\}$ satisfying the following properties:

- for all $e, e' \in E(CG)$ such that e connects to e', we have $\delta(e) \leq \delta(e')$, and
- for all $e \in E(CG)$ and $l \in L$, if ACQ(e, l) and $\delta(e) < \infty$, we have

$$\big|\{e' \in E(CG) : \text{ACQ}(e', l) \text{ and } \delta(e') < \delta(e)\}\big| = \big|\{e' \in E(CG) : \text{REL}(e', l) \text{ and } \delta(e') < \delta(e)\}\big| \quad (1)$$

At time $t = 1$, we have $\delta(e) = 1$ for the single edge in G_0. A $(t+1)$-step execution δ' is an ***extension*** of a t-step execution δ if for all $e \in E(CG)$ such that $\delta(e) \leq t$, we have $\delta'(e) = \delta(e)$. A t-step execution of CG is ***complete*** if $|CG| = t$ and ***incomplete*** otherwise. If $\delta(e) = \infty$ and there is no extension δ' of δ such that $\delta'(e) < \infty$ then e ***cannot be executed*** δ

6. DEADLOCK FREEDOM

In this section, we prove that the progressive HCC protocol is deadlock free. We define a computation graph CG over a strictly linearly ordered set $(L, >)$ of locks and a set Z of memory locations to be deadlock free if all incomplete t-step executions of CG can be extended. We show that legal computations can be extended and that HCC produces a legal computation graph. We conclude that HCC is deadlock free.

Lemma 1. *Any legal computation graph CG over a strictly linearly ordered set $(L, >)$ of locks and set Z of memory locations is deadlock free.*

PROOF. Suppose that δ is an incomplete t-step execution of CG that cannot be extended ($t < |CG|$). Define $K = \{e \in E(CG) : e$ connects to $e' \in E(CG)$ and $\delta(e') = \infty\}$. Let $L^{(t)} = \{l \in L :$ there exists $e \in K$ where e holds $l\}$ be the set of locks that are being held by the edges in K. Let $l^* = \max(L^{(t)})$ according to the lock ordering, and let $X = \{e \in K : e \text{ holds } l^*\}$.

Since all edges $e \in X$ hold l^* and CG is a legal computation graph, there exist $e_1, e_2 \in E(G)$ such that ACQ(e_1, l^*), $\delta(e_1) \leq t$ and REL(e_2, l^*), $\delta(e_2) = \infty$, and $e \in X$ implies $e \in \mathrm{CR}(e_1, e_2, l^*)$. Observe that $e \neq e_2$, since $\delta(e_2) = \infty$ and $\delta(e) \leq t$.

Let $e \in X$ be the edge such that for all $e' \in X$, the longest path from tail(e) to ender(e_2) is at least as long as the longest path from tail(e') to $ender(e_2)$. Assume that e belongs to an activity $G \in CG$. Let us examine the properties of e and obtain a contradiction to the supposition that δ cannot be extended.

If head(e) is a sink node, then $e \notin \mathrm{CR}(e_1, e_2, l^*)$, because it cannot be on any path between tail(e_1) and ender(e_2). Therefore, e cannot hold lock l^*, contradicting the fact that e holds l^*.

Suppose that head(e) is a join node and there exists an edge $y \neq e \in E(CG)$ such that head(y) = head(e) and $\delta(y) = \infty$. By the definition of a critical region, e_1 must lie on all paths from root(G) to ender(e_2). Because CG is a legal computation in which all activities are well formed and head(y) = head(e), we have that y holds l^*. Since head(y) = head(e), edge e cannot lead to y, and therefore e and y cannot belong to the same path that leads to ender(e). Therefore, there must be an edge $e' \in E(CG)$ such that $e' \in \mathrm{CR}(e_1, e_2, l^*)$, $e' \in X$, and y belongs to a path from tail(e') to ender(e_2). But, then the path from tail(e') to ender(e_2) includes at least one more edge (y) than the path from tail(e) to ender(e_2), contradicting the way e was chosen.

If head(e) is neither a sink nor a join, or is a join but every edge $y \neq e$ such that head(e) = head(y) has $\delta(y) \leq t$. Then there exists an $e' \in E(CG)$ such that e connects to e', in-deg(head(e')) = 1, and e' cannot be executed. If for all such e' and for all $l \in L$, we have $\neg$ACQ(e', l), then by the t-step execution definition, δ can be extended to $\delta' : E(CG) \to \{1, 2\,..\,t, t+1\} \cup \{\infty\}$ such that $\delta'(e') = t + 1$ is a $t + 1$-step execution, contradicting the assumption that δ cannot be extended.

Otherwise, let $D = \{e' \in E(CG)$ such that e connects to e' and ACQ(e', l) for some lock $l \in L$ and e' cannot be executed$\}$. There exists $e'' \in D$ such that e'' holds l^*, for otherwise e cannot be part of any critical region, contradicting the assumption that CG is a legal computation graph. Since e'' cannot be executed, it follows that Equation (1) does not hold, and we must consider two cases:

- If $|\{e' \in E(CG) :$ ACQ(e', l) and $\delta(e') < \delta(e)\}| < |\{e' \in E(CG) :$ REL(e', l) and $\delta(e') < \delta(e)\}|$, then there must exist an edge $e_2 \in E(CG)$ such that REL(e_2, l) and $\delta(e_2) \leq t$ and there is no $e_1 \in CG$ such that ACQ(e_1, l) and a critical region $\mathrm{CR}(e_1, e_2, l)$ is formed. Hence, CG contains an activity that is not well structured, contradicting the fact the CG is a legal computation.
- If $|\{e' \in E(CG) :$ ACQ(e', l) and $\delta(e') < \delta(e)\}| > |\{e' \in E(CG) :$ REL(e', l) and $\delta(e') < \delta(e)\}|$, then if $l \leq l^*$, in which case e'' is acquiring a lower-ordered lock than it already holds, contradicting the assumption that all activities in CG respect the order of locks. If $l > l^*$ and there is at least one

more acquire of l than releases, it follows that l is held by some $s \in K$, and thus s holds the largest lock in K, contradicting the the maximality of l^*

Hence, δ can be extended, and CG is deadlock free. □

The order of locks in HCC

The strict linear ordering of locks in HCC is defined with respect to the level of the caches in the cache hierarchy. If i is lower than cache j, and l_i is a line in cache i and l_j is a line in cache j, then the order of the locks is the following:

1. $l_i < \text{QWIRE}\langle i, j\rangle < l_j$,
2. $l_j < \text{WIRE}\langle j, i\rangle$,
3. $l_i < \text{YWIRE}\langle i, j\rangle$,
4. $\text{WIRE}\langle j, i\rangle < \text{YWIRE}\langle i, j\rangle$,
5. $\text{EWIRE}\langle i\rangle < l_i$.

While processing a message, a cache j may need to acquire outgoing wires. The order of the locks on the outgoing wires from j to parent p, Child 0 $c0$, and Child 1 $c1$ is as follows:

1. $\text{YWIRE}\langle j, p\rangle < \text{QWIRE}\langle j, p\rangle$,
2. $\text{QWIRE}\langle j, p\rangle < \text{WIRE}\langle j, c0\rangle < \text{WIRE}\langle j, c1\rangle$.

The partial order defined by these constraints can be extended to a strict linear order.

We now show that all the activities formed by the HCC program acquire locks in the proper order.

Lemma 2. *All activities in CG are legal.*

PROOF. A wire lock is released after all the resources needed to process the message it delivered are acquired. A line lock is acquired when the request on this line is processed and is held until the reply is processed. Join edges connect the end of a joiner to the first instruction of the handler invoked by the last updater. Therefore, when the activity graph is built, it is well structured. Following the code in Section 4, one can verify that every ACQUIRE obtains a lock ordered higher than what it already holds. Although it is possible for a handler to execute TRY-ACQUIRE on a lower-ordered lock, the handler never waits for this lock — if the lock cannot be acquired, the handler simply proceeds. Therefore, the activities respect the order of locks and are legal. □

Theorem 3. *The HCC protocol is deadlock free.*

PROOF. Follows from Lemma 2. □

7. SEQUENTIAL CONSISTENCY

Our definition of sequential consistency follows the definitions in [15] with some minor but salient differences. We describe our definition and conditions for sequential consistency, and we prove that computation graphs produced by the HCC protocol satisfy these conditions.

Let $V_0(CG) = V(CG) - \{s_0\}$, and let $E_0(CG) = E(CG) - E(G_0)$. An ***observer function*** $\Phi : E_0(CG) \times Z \to E(CG)$ on a computation graph CG satisfies the following properties:

1. For all $z \in Z$ and $e \in E_0(CG)$, we have $\text{W}(\Phi(e, z), z)$.
2. For all $z \in Z$ and $e \in E(CG)$, we have $e \not\preceq \Phi(e, z)$.

A ***memory model*** over a set Z of memory locations is a pair (CG, Φ), where CG is a computation graph and Φ is an observer function for CG. The observer function is defined for all memory locations and all edges. For our purposes, we only require a relaxed version of the observer function which is defined for all edges on one memory location. A ***concrete (partial) observer function*** $\Phi : E(CG) \times Z \to E(CG)$ satisfies the following constraints:

- For all $e \in E_0(CG)$ and $z \in Z$ with $\text{RW}(e, z)$, there exists an $e' \neq e \in E(CG)$ such that $\text{W}(e', z)$ and $\Phi(e, z) = e'$.
- For all $e \in E_0(CG)$, and $z_1, z_2 \in Z$ such that $z_1 \neq z_2$ and $\text{RW}(e, z_1)$, the value of $\Phi(e, z_2)$ is undefined.

Let $e, e' \in E(CG)$. Given a concrete observer function Φ on a computation graph CG, define the set of ***dependency edges*** to be $DE(CG, \Phi) = \{(head(e), head(e')) : \text{there exists } z \in Z \text{ such that } e = \Phi(e', z)\}$. The set of ***antidependency edges*** is $AE(CG, \Phi) = \{(\text{head}(e), \text{head}(e')) : \text{there exists } z \in Z \text{ such that } \text{R}(e, z), \text{W}(e', z), \text{ and } \Phi(e, z) = \Phi(e', z)\}$ A concrete observer function Φ satisfies the ***chained-writers property*** if for all $z \in Z$, the subgraph of dependency edges induced by $\text{W}(E, z) \subseteq E(CG)$ forms a chain. That is, $e, e' \in \text{W}(E, z)$ where $e \neq e'$ implies that $\Phi(e, z) \neq \Phi(e', z)$.

Definition 1. *Let CG be a computation, and let τ be a topological sort of $E(CG)$. The **last writer** according to τ is the function $\mathcal{L} : E_0(CG) \times Z \to E(CG)$ such that for all $e \in E(CG)$ and $z \in Z$, we have*

1. $\text{W}(\mathcal{L}(e, z), z)$,
2. $\tau(\mathcal{L}(e, z)) < \tau(e)$,
3. *for all $e' \in E(CG)$ such that $\text{W}(e', z)$, we have $\tau(e') \leq \tau(\mathcal{L}(e, z))$ or $\tau(e) \leq \tau(e')$.*

The function $\mathcal{L}$ is an observer function, because the first properties of both $\mathcal{L}$ and an observer function are identical and $\mathcal{L}$'s second property — $\tau(\mathcal{L}(e, z)) < \tau(e)$ — implies the second property of an observer function — $e \not\preceq \Phi(e, z)$.[2] A memory model $\{CG, \Phi\}$ over a set Z of memory locations is defined to be ***sequentially consistent*** ($\mathcal{SC}$) if CG is a computation graph and Φ is a last-writer function for CG.

Lemma 4. *Let Φ be a concrete observer function on a computation graph CG that satisfies the chained-writers property, and suppose that the augmented computation graph CG' defined by $V(CG') = V(CG)$ and $E(CG') = E(CG) \cup DE(CG, \Phi) \cup AE(CG, \Phi)$ is acyclic. Then, Φ can be extended to a last-writer observer function on CG such that $(CG, \Phi) \in \mathcal{SC}$.*

PROOF. For convenience, define $V = V(CG) = V(CG')$ and $E = E(CG) = E(CG')$. Since CG' is acyclic, there exists a topological sort τ of E. Let $\mathcal{L}$ be the last-writer function according to τ. We shall show that $\Phi(e, z) = \mathcal{L}(e, z)$ for all $z \in Z$ and $e \in E$ for which $\Phi(e, z)$ is defined.

Assume for the purpose of contradiction that for some $z \in Z$, there exists an $e' \in E$ for which $\Phi(e', z) \neq \mathcal{L}(e', z)$. Let $e \in \{e' \in V : \Phi(e', z) \neq \mathcal{L}(e', z)\}$ such that $\tau(e) < \tau(e'')$ for all $e'' \in \{e' \in V : \Phi(e', z) \neq \mathcal{L}(e', z)\}$, and let $e_1 = \mathcal{L}(e, z)$ and $e_2 = \Phi(e, z)$. Thus, by assumption, we have $e_1 \neq e_2$. Moreover, by the definitions of a concrete observer function and Definition 1, it must be that $\text{W}(e_1, z)$, $\text{W}(e_2, z)$, $\tau(e_1) < \tau(e)$, and $\tau(e_2) < \tau(e)$. We consider two cases:

- $\tau(e_1) < \tau(e_2) < \tau(e)$: From Property 3 of Definition 1, we have $\tau(e_2) \leq \tau(\mathcal{L}(e, z))$ or $\tau(e) \leq \tau(e_2)$, but since $\tau(e_2) < \tau(e)$, it follows that $\tau(e_2) \leq \tau(\mathcal{L}(e, z))$. Since $e_1 = \mathcal{L}(e, z)$, we obtain the contradiction that $\tau(e_2) \leq \tau(e_1)$.
- $\tau(e_2) < \tau(e_1) < \tau(e)$: Let $k \in E$ be the edge with minimum $\tau(k)$ such that $\tau(e_2) < \tau(k)$ and $\text{W}(k, z)$. Since $\text{W}(e_1, z)$ holds, we have that $\tau(k) \leq \tau(e_1)$, and hence we have $\tau(e_2) < \tau(k) \leq \tau(e_1) < \tau(e)$.

[2] Frigo's original definition allows the second property of a last writer to be $\tau(\mathcal{L}(e, z)) \preceq \tau(e)$. We restrict that a write operation cannot observe itself, but rather observes a previous write, as if the write operation first performs a read. As shown, $\mathcal{L}$ is still an observer function, since if $\tau(\mathcal{L}(e, z)) < \tau(e)$, we have $e \not\preceq \Phi(e, z)$.

By Definition 1, we have $e_2 = \mathcal{L}(k, z)$. Since $\tau(e)$ is the minimum over all other $e' \in E$ for which $\mathcal{L}(e', z) \neq \Phi(e', z)$, it must be that $e_2 = \Phi(e, z) = \Phi(k, z)$. But, since $k \neq e$ and Φ satisfies the chained-writer property, we obtain a contradiction.

Hence, Φ can be extended to a last-writer observer function on CG such that $(CG, \Phi) \in \mathcal{SC}$. □

Lemma 5. *The HCC protocol preserves the MSI property.*

PROOF. Follows the transition function and is omitted. □

To prove that HCC implements sequential consistency, we must show that it satisfies the conditions of Lemma 4. Specifically, adding dependency and antidependency edges to CG must leave CG acyclic, and we must exhibit a concrete observer function Φ on CG that satisfies the chained-writers property. Since HCC is deadlock free by Theorem 3, we can presume that the execution is complete.

In order to define the dependency and antidependency edges, we include as part of the state of each message m a ***writer*** field $m.writer \in E(CG)$ and an invalidation list $m.inval \subseteq E(CG)$, and similarly we include for each cache line l the fields $l.writer$ and $l.inval$. This information is not needed for the actual operation of HCC and is only added for the purpose of proving sequential consistency. Intuitively, the writer is the edge (instruction) that wrote the current value to the corresponding location, and the invalidation list contains edges that change state of a line in an L1-cache to Invalid. These values are updated as follows:

- Whenever a cache receives a message with a given writer, it updates the writer field in corresponding cache line.
- Whenever an instruction e executes in a handler operating in a cache i such that $\mathrm{W}(e, z)$ for some location z, we set $e.writer \leftarrow$ CACHE-LINE$(i, z).writer$ and CACHE-LINE$(i, z).writer \leftarrow e$.
- Whenever a cache receives a message with an invalidation list, it adds all the edges in the list to the existing invalidation list of corresponding cache line.
- Whenever Line 7 of PARENT-REQ$(from, to)$ executes, thereby causing $\mathrm{RW}(e, z)$ to hold for a location z, then e is added to CACHE-LINE$(to, z).inval$ if the state of the line is changed to: CACHE-LINE$(to, z).LS = I$.
- Whenever Line 6 of LOAD$(from, z')$ or STORE$(from, z')$ executes where if CACHE-LINE$(from, z')$ is already occupied by location z, thereby causing $\mathrm{RW}(e, z)$ to hold for a location $z \neq z'$, then e is added to CACHE-LINE$(from, z').inval$ and the line state is changed to CACHE-LINE$(from, z').LS = I$.

We now add dependency and antidependency edges as follows:

1. For all $e' \in E(CG)$ such that $\mathrm{R}(e', z)$, if $e = e'.writer$, then a dependency edge $(\mathrm{head}(e), \mathrm{head}(e'))$ is added to $E(CG)$.
2. For all $e' \in E(CG)$ such that $\mathrm{W}(e', z)$, if $e = e'.writer$, then a dependency edge $(\mathrm{head}(e), \mathrm{head}(e'))$ is added to $E(CG)$.
3. For all $e' \in E(CG)$ such that $\mathrm{W}(e', z)$ then for all e that are in the invalidation list attached to the state of the line, we add the antidependency edge $(\mathrm{head}(e), \mathrm{head}(e'))$ to $E(CG)$.

Lemma 6. *The computation graph CG with dependency and antidependency edges is acyclic.*

PROOF. Control edges that follow the program order cannot create cycles in CG. By the way the $writer$ and $inval$ information is gathered, from Lemma 5, and by the way the dependency and antidependency edges $(\mathrm{head}(e), \mathrm{head}(e'))$ are added, it must be that and $\tau(e) < \tau(e')$. Therefore, the computation graph CG is acyclic. □

The partial function $\mathcal{L}_{\mathcal{HCC}} : E(CG) \times Z \rightarrow E(CG)$ is defined such that for all $e \in E(CG)$ and $z \in Z$, we have that $e' = \mathcal{L}_{\mathcal{HCC}}(e)$ if the following constraints are satisfied: (1) $\mathrm{RW}(e, z) = \mathrm{RW}(e', z)$, and (2) $e' = writer(e)$. If no such e' exists, then $\mathcal{L}_{\mathcal{HCC}}(e)$ is undefined.

Lemma 7. *The partial function $\mathcal{L}_{\mathcal{HCC}}$ is a concrete observer function that satisfies the chained-writer property.*

PROOF. The partial function $\mathcal{L}_{\mathcal{HCC}}$ satisfies the first property of a concrete observer function, since e and e' access the same memory location and by the way CG was constructed. For $e' = \mathcal{L}_{\mathcal{HCC}}(e)$ to hold requires both e and e' to access the same memory location. Therefore, $\mathcal{L}_{\mathcal{HCC}}$ satisfies the second property of a concrete observer function as well. It also satisfies the chained-writers property, since by Lemma 5, there is only one L1-cache that can update a memory location at a given time. □

Theorem 8. *HCC implements sequential consistency.*

PROOF. Lemma 6 shows that CG together with its dependency and antidependency edges is acyclic. Lemma 7 proves that $\mathcal{L}_{\mathcal{HCC}}$ is a concrete observer function that satisfies the chained-writer property. Since the processors follow the order-consumption property, HCC implements sequential consistency. □

8. RELATED WORK

Researchers have studied consistency protocols for large-scale multiprocessors that employ hierarchical shared caches since the late 1980's. Wilson [40] proposes a tree-like hierarchy of shared caches based on buses and sketches an adaptation of Goodman's consistency protocol [16]. Cheriton *et al.* [12] describe a distributed parallel multi-computer that uses a memory hierarchy based on shared caches. Mizrahi *et al.* [30] outline a distributed directory protocol in a system where memory is embedded in the switches of an interconnection network. Their protocol restricts sharing of the data, however, allowing only one copy of each memory location to reside in the system. Yang *et al.* [41] describe a consistency protocol that works on both the Wilson and Mizrahi *et al.* architectures. None of this research, however, addresses issues of deadlock and message races that result from the hierarchy. These protocols employ the inclusion property, and in [7], several conditions for imposing the inclusion property for fully and set-associative cache hierarchies are presented. Recently, attention has turned to architectures that support CMP's with variable degrees of hierarchy [1–3,8,9,21,29,35,38]. Acacio *et al.* [3] propose a hierarchical scheme for directory-based consistency using a multilayer clustering concept. Shen and Arvind [36] offer a progressive protocol based on term rewriting systems.

Some researchers [26, 31, 37] have suggested consistency protocols for limited-height memory hierarchies. Bolotin *et al.* [8] describe an architecture with two levels of cache in which the L2-cache stalls to provide a serialization point. They show how to improve performance of a directory-based consistency protocol using a priority-based network-on-chip. Marty and Hill [29] propose a two-level virtual-memory hierarchy that outperforms a flattened architecture. Acacio [2] suggests a three-level directory architecture, while Chang and Sohi [9] offer a cooperative caching protocol.

Several studies on interconnects on CMP's [11, 13, 20, 21] propose a variety of interconnects and suggest ways to reduce consistency traffic. Martin *et al.* [27] describe a token-based consistency protocol for a ring-based architecture. Marty and Hill [28] also describe a protocol for a ring-based architecture, focusing mainly on efficiently and correctly ordering requests. Strauss *et al.* [39] suggest a method to improve snooping protocols in ring-based CMP's.

Some researchers [10,19,32] have proposed protocols for hierarchies of caches that share the same memory location, but the caches

themselves are not shared. This restriction helps in the invalidation process, but in most cases read and write requests must still go through memory. Eisley *et al.* [14] propose an architecture in which directories are embedded within routing nodes. To address the deadlock situation that might arise when a line is invalidated, their scheme uses timeouts.

9. CONCLUSIONS AND FUTURE WORK

Cache-consistency protocols developed for hierarchical shared caches often introduce nondeterminacy into the system by using timeouts. Moreover, many existing protocols have serialization bottlenecks that induce latency greater than the network diameter. The HCC framework offers a progressive and scalable strategy for implementing fully scalable cache-consistency architectures and protocols that overcome these deficiencies.

This work suggests numerous extensions. The transition tables can be compacted and logic simplified. The switches in caches can be parallelized to handle multiple independent messages at one time. Variants on the MSI strategy, such as MESI and MOESI, can be implemented using the HCC framework. Caches can have a larger number of children or parents than the binary scheme presented, and irregular network structures can be supported.

Interesting avenues of research include investigating the impact of relaxing the strong inclusion and unique-path properties. In particular, what is the tradeoff between storage and communication if only a weak-inclusion property is enforced? Do progressive protocols exist for networks such as meshes where messages may have several paths along which to travel?

10. REFERENCES

[1] M. Acacio, J. Gonzalez, J. Garcia, and J. Duato. A two-level directory architecture for highly scalable cc-NUMA multiprocessors. *Trans. on Parallel and Distributed Systems*, 16(1):67–79, 2005.

[2] M. E. Acacio. An architecture for high-performance scalable shared-memory multiprocessors exploiting on-chip integration. *IEEE Trans. on Parallel Distributed Systems*, 15(8):755–768, 2004.

[3] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. A new scalable directory architecture for large-scale multiprocessors. In *IEEE HPCA*, 2001.

[4] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*, July 2007.

[5] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *ISCA*, 2008.

[6] Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification. In *MEMOCODE*, 2003.

[7] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *ISCA*, 1988.

[8] E. G. Bolotin, Z. Cidon, I. Ginosar, and A. R. Kolodny. The power of priority: NoC based distributed cache coherency. In *NOCS*, 2007.

[9] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *ISCA*, 2006.

[10] Y. Chang and L. Bhuyan. An efficient tree cache coherence protocol for distributed shared memory multiprocessors. *IEEE Trans. on Computers*, 48(3):352–360, 1999.

[11] L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J. B. Carter. Interconnect-aware coherence protocols for chip multiprocessors. In *ISCA*, 2006.

[12] D. R. Cheriton, H. A. Goosen, and P. D. Boyle. Multi-level shared caching techniques for scalability in VMP-M/C. In *ISCA*, pages 16–24, 1989.

[13] W. J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *DAC*, 2001.

[14] N. Eisley, L.-S. Peh, and L. Shang. In-network cache coherence. In *MICRO*, 2006.

[15] M. Frigo. The weakest reasonable memory model. Master's thesis, MIT EECS, 1998.

[16] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *ISCA*, 1983.

[17] M. Hill. Multiprocessors should support simple memory consistency models. *IEEE Computer*, 31(8):28–34, 1998.

[18] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, October 2006.

[19] S. Kaxiras and J. R. Goodman. The GLOW cache coherence protocol extensions for widely shared data. In *ICS*, 1996.

[20] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS-X*, 2002.

[21] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling. In *ISCA*, 2005.

[22] C.-Y. Lam and S. E. Madnick. Propeties of storage hierarchy systems with multiple page sizes and redundant data. *ACM Transaction on Database Systems*, 4(3):345–367, 1979.

[23] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programm. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.

[24] F. T. Leighton and B. M. Maggs. Fast algorithms for routing around faults in multibutterflies and randomly-wired splitter networks. *IEEE Transaction on Computers*, 41(5):578–587, 1992.

[25] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. In *SPAA*, 1992.

[26] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: implementation and performance. In *ISCA*, 1998.

[27] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: decoupling performance and correctness. In *ISCA*, 2003.

[28] M. R. Marty and M. D. Hill. Coherence ordering for ring-based chip multiprocessors. In *MICRO*, 2006.

[29] M. R. Marty and M. D. Hill. Virtual hierarchies to support server consolidation. In *ISCA*, 2007.

[30] H. E. Mizrahi, J. L. Baer, E. D. Lazowska, and J. Zahorjan. Introducing memory into the switch elements of multiprocessor interconnection networks. In *ISCA*, 1989.

[31] S. Mori, H. Saito, M. Goshima, S. Tomita, M. Yanagihara, T. Tanaka, D. Fraser, K. Joe, and H. Nitta. A distributed shared memory multiprocessor ASURA: memory and cache architecture. In *Supercomputing*, 1993.

[32] H. Nilsson and P. Stenstrom. The scalable tree protocol-a cache coherence approach for large-scale multiprocessors. In *IPDPS*, 1992.

[33] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, second edition, 1998.

[34] S. Przybylski, M. Horowitz, and J. Hennessy. Characteristics of performance-optimal multi-level cache hierarchies. In *ISCA*, 1989.

[35] A. Ros, M. E. Acacio, and J. M. Garcia. An efficient cache design for scalable glueless shared-memory multiprocessors. In *Computing Frontiers*. ACM Press, 2006.

[36] X. Shen and Arvind. Specification of memory models and design of provably correct cache coherence protocols. Technical Report 398, MIT Computation Structures Group, 1997.

[37] X. Shen, Arvind, and L. Rudolph. CACHET: an adaptive cache coherence protocol for distributed shared-memory systems. In *ICS*, 1999.

[38] E. Speight, H. Shafi, L. Zhang, and R. Rajamony. Adaptive mechanisms and policies for managing cache hierarchies in chip multiprocessors. In *ISCA*, 2005.

[39] K. Strauss, X. Shen, and J. Torrellas. Flexible snooping: adaptive forwarding and filtering of snoops in embedded-ring multiprocessors. *ISCA*, 2006.

[40] A. W. Wilson, Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *ISCA*, 1987.

[41] Q. Yang, G. Thangadurai, and L. M. Bhuyan. Design of an adaptive cache coherence protocol for large scale multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 3(3):281–293, 1992.