1 Appendix

This appendix includes our complete proofs of correctness that MemGlue$_O$ and MemGlue$_U$ uphold the C11 MCM, in §1.1 and §1.2 respectively. Next, we include our complete transition tables for MemGlue$_O$ and MemGlue$_U$ in §1.3. Then, a diagram of the complete design for MemGlue$_U$'s consistency controller (§1.4) is shown. Finally, since MemGlue's functionality relies on many different counters, we provide an outline for preemptively resetting the system to avoid counter overflow in §1.5.

1.1 Proof of Correctness: Ordered MemGlue

Proof Goal: for each C11 axiom, if the axiom forbids an execution of a program, then this execution is not observable in MemGlue.

1.1.1 Preliminaries.

Definition 1.1. *A cache line in a cluster LLC (tracked by the cluster's shim) or the CC is* updated *when a* RRESP *(cluster LLC), cluster-local write (cluster LLC), or* WRITE *(cluster LLC or CC) modifies the cache line's data. Throughout the proof, we case on these three types of updates.*

Definition 1.2. $\prec_{CC}$ *is the order in which* WRITE*s are processed at the CC.*

Definition 1.3. *The* write message (WRITE) associated with an update U *at shim* S *is defined as follows:*

- *If* U *is a local write, its* associated write message *is the* WRITE *sent by* S *to the CC when* U *is processed by* S.
- *If* U *is a* WRITE*, its associated write message is the original* WRITE *sent to the CC by another shim* $S' \neq S$ *which triggered* U *to be sent to* S.
- *If* U *is a* RRESP*, its associated write message is the* WRITE *to the CC whose written data is contained in* U.

Notation. U.ts *denotes the* timestamp associated with some update $U \in \{RRESP, WRITE\}$*, i.e., the timestamp assigned by the CC to the update's associated* WRITE*. Recall that* U *is tagged with this CC-assigned timestamp when it is sent from the CC (§??).*
Notation. *By* $U_i \prec_{CC} U_j$*, we denote that the associated* WRITE*s of updates* U_i *and* U_j *are ordered by* $\prec_{CC}$*. By* $U_i \preceq_{CC} U_j$*, we denote that the associated* WRITE*s of* U_i *and* U_j *may also be the same.*
Notation. shim[S] *denotes the metadata shadow cache tracked by shim* S *(§??).*

Definition 1.4. shim[S][x] *denotes the metadata associated with the cache line containing address* x *tracked by shim* S*. Without loss of generality, in this proof we assume cache line-granularity accesses, so* x *is a cache line address.*

- shim[S][x].wc_non_local *denotes the* non-local write count *for the cache line containing address* x *tracked by shim* S*, defined as the number of* WRITE*s to* x received *by* S *(whether or not each* WRITE*'s data was explicitly written to the LLC of* S*'s cluster) since* S *most recently became a sharer of* x*.*
- shim[S][x].wc_local *denotes the* write count *for the cache line containing address* x *at shim* S*, defined as the number of local writes to address* x *performed at the cluster associated with* S *since it most recently became a sharer of* x*.*
- shim[S][x].ts_{sync} *denotes the* synchronizing timestamp *of the cache line containing address* x *tracked by shim* S*, defined as the initial timestamp value written to* shim[S][x].ts *when that cache line most recently transitioned from state* I *to* V*. Per the MemGlue$_O$ protocol, a cache line transitions from* I *to* V *on a read miss after sending/receiving a* RREQ/RRESP*, or on a write miss. Thus,* ts_{sync} *will either be:*
  1. *the timestamp assigned to some write* W *to* x *(§??), where* W *is read from by a read* R *that misses at shim* S *(via a* RREQ/RRESP *pair, §??), or*
  2. *the timestamp assigned to the last write to* x *performed at the CC before* S *synchronized with the CC, returned in a* WRITE_ACK *message after a local write performs and makes* S *a sharer of* x *(§??).*
- shim[S][x].W_{sync} *denotes the* synchronizing write*, defined as the write whose timestamp was assigned to* shim[S][x].ts_{sync}.

Lemma (Non-Local Writes Increment the Timestamp): For any shim S that shares (tracks) some address x, any WRITE message to x that *arrives* at S will increment shim[S][x].ts by 1 upon being *processed* by S.

Proof (Non-Local Writes Increment the Timestamp). We must prove this lemma explicitly because in the MemGlue specification, WRITEs which pass the timestamp check write their timestamps to shim[S][x].ts instead of incrementing the previous shim[S][x].ts value. In this case, we must prove that this action is equivalent to incrementing shim[S][x].ts.

We proceed by induction on the number of non-local WRITEs that arrive to S.

Base Case: Suppose S synchronizes with the CC and becomes a sharer of x on some write $\texttt{shim[S][x].W}_{\texttt{sync}}$ with timestamp $\texttt{shim[S][x].ts}_{\texttt{sync}}$, and then S immediately processes $L \geq 0$ local writes followed by one non-local WRITE $\texttt{W}_{\texttt{S}'}$ (i.e., a write originating from any shim $\texttt{S}' \neq \texttt{S}$). We case on the ordering of these (local and non-local) writes at the CC.

Case 1: $\texttt{W}_{\texttt{S}'}$ arrived at the CC after all L local writes. Then $\texttt{W}_{\texttt{S}'}\texttt{.ts}$ is $\texttt{shim[S][x].ts}_{\texttt{sync}} + L + 1$. This follows from Case 1 and our network ordering assumption. In particular, since $\texttt{W}_{\texttt{S}'}$ was the first non-local write to arrive at S after S became a sharer of x, network ordering guarantees that it was also the first non-local WRITE to arrive at the CC after S became a sharer of x. Also, $\texttt{shim[S][x].ts} = \texttt{shim[S][x].ts}_{\texttt{sync}} + L$ upon $\texttt{W}_{\texttt{S}'}$'s arrival to S, since local writes increment shim[S][x].ts by 1. Then, $\texttt{W}_{\texttt{S}'}$ will be allowed to write its data to S's associated LLC because its timestamp is greater than shim[S][x].ts, and it will also write its timestamp, $\texttt{shim[S][x].ts}_{\texttt{sync}} + L + 1$, to shim[S][x].ts, per MemGlue's specification (§??). Since the previous value of shim[S][x].ts was $\texttt{shim[S][x].ts}_{\texttt{sync}} + L$, this means that $\texttt{W}_{\texttt{S}'}$ increments the prior timestamp by 1.

Case 2: $\texttt{W}_{\texttt{S}'}$ arrived at the CC after L_{pre} of the L local writes, and L_{post} of the local writes arrived at the CC after $\texttt{W}_{\texttt{S}'}$. So $L = L_{pre} + L_{post}$, and $L_{post} > 0$. We know that the CC will set $\texttt{W}_{\texttt{S}'}\texttt{.ts}$ to be $\texttt{shim[S][x].ts}_{\texttt{sync}} + L_{pre} + 1$ (by the same reasoning in Case 1), and that shim[S][x].ts will be $\texttt{shim[S][x].ts}_{\texttt{sync}} + L_{pre} + L_{post}$ upon the arrival of $\texttt{W}_{\texttt{S}'}$. Since $L_{post} > 0$, then $\texttt{W}_{\texttt{S}'}\texttt{.ts}$ will fail the timestamp check, and it will not write its data to S's associated LLC. However, it will increment shim[S][x].ts by 1 (§??).

Therefore, in both cases, $\texttt{W}_{\texttt{S}'}$ causes shim[S][x].ts to be incremented by 1.

Induction Hypothesis: Suppose N non-local writes to x have been processed at S, and each has incremented shim[S][x].ts by 1.

Suppose L_{pre} local writes performed before the N^{th} non-local write, and L_{post} local writes after. Since local writes increment the timestamp by 1, and the N previous non-local writes incremented the timestamp by 1, then $\texttt{shim[S][x].ts} = \texttt{shim[S][x].ts}_{\texttt{sync}} + N + L_{pre} + L_{post}$.

Induction Step: Consider the $N + 1^{\text{th}}$ non-local write $\texttt{W}_{\texttt{S}'}$.

Case 1: $\texttt{W}_{\texttt{S}'}$ arrived at the CC after all L_{post} local writes which occurred after the last of the N non-local writes. Then $\texttt{W}_{\texttt{S}'}\texttt{.ts}$ is set to $\texttt{shim[S][x].ts}_{\texttt{sync}} + N + L_{pre} + L_{post} + 1$ by the CC. When it arrives to S, its timestamp is one greater than shim[S][x].ts, so it effectively increments shim[S][x].ts by 1.

Case 2: $\texttt{W}_{\texttt{S}'}$ arrived at the CC before some of the L_{post} local writes. Say L_{post_0} arrived before $\texttt{W}_{\texttt{S}'}$, and L_{post_1} arrived after, and $L_{post\_1} > 0$. Then, $\texttt{W}_{\texttt{S}'}\texttt{.ts}$ is $\texttt{shim[S][x].ts}_{\texttt{sync}} + N + L_{pre} + L_{post\_0} + 1$ by the CC. And $\texttt{shim[S][x].ts} = \texttt{shim[S][x].ts}_{\texttt{sync}} + N + L_{pre} + L_{post\_0} + L_{post\_1}$, which is greater than or equal to $\texttt{W}_{\texttt{S}'}\texttt{.ts}$ since $L_{post\_1} > 0$. Therefore, $\texttt{W}_{\texttt{S}'}$ does not write its data, but it does increment the timestamp by 1 (§??).

Therefore, by induction, we have shown that every non-local WRITE to x that arrives at S increments shim[S][x].ts by 1.

Lemma (Timestamp Invariant) For any shim S and address x that S tracks, $\texttt{shim[S][x].ts} = \texttt{shim[S][x].ts}_{\texttt{sync}} + \texttt{shim[S][x].wc\_local} + \texttt{shim[S][x].wc\_non\_local}$.

Proof (Timestamp Invariant). Consider an arbitrary shim S and address x that S tracks, and suppose that when S synchronizes with the CC to become a sharer of x, it is supplied with timestamp $\texttt{shim[S][x].ts}_{\texttt{sync}}$. By Lemma (Non-Local Writes Increment the Timestamp), non-local WRITEs increment shim[S][x].ts by 1. By Definition 1.4, the number of non-local WRITEs to x at S is tracked by shim[S][x].wc_non_local. By the protocol specification (§??), all local writes increment shim[S][x].ts by 1, and by Definition 1.4 the number of local writes to x at S is tracked by shim[S][x].wc_local. Therefore, $\texttt{shim[S][x].ts} = \texttt{shim[S][x].ts}_{\texttt{sync}} + \texttt{shim[S][x].wc\_local} + \texttt{shim[S][x].wc\_non\_local}$.

Lemma (update order): For any shim S, address x tracked by S, and pair of updates $\texttt{U}_\texttt{i}, \texttt{U}_\texttt{j}$ to shim[S][x] such that $\texttt{U}_\texttt{i}$ updates shim[S][x] before $\texttt{U}_\texttt{j}$, $\texttt{U}_\texttt{i} \preceq_{\texttt{CC}} \texttt{U}_\texttt{j}$.

Proof (update order). Consider arbitrary shim S and address x tracked by S. We show by induction on the sequence of updates to shim[S][x] that these updates are ordered by $\preceq_{\texttt{CC}}$.

Base Case. Consider $\texttt{U}_0$ and $\texttt{U}_1$, the first and second updates to x that update shim[S][x] after S has become a sharer of x for the first time. We first case on $\texttt{U}_0$.

Case 1: $\texttt{U}_0$ is a RRESP. We now case on $\texttt{U}_1$.

- Sub-Case A: $\texttt{U}_1$ is a RRESP. Then, x was evicted between updates $\texttt{U}_0$ and $\texttt{U}_1$. Otherwise, the read that triggered $\texttt{U}_1$ would have read its value from a local cache (via a hit) instead of sending a RREQ to the CC. Each RRESP contains the data and timestamp for its specified address that exist in the CC at the time the CC processes its corresponding RREQ. The RRESPs $\texttt{U}_0$ and $\texttt{U}_1$ arrive to S in the order that they were sent from the CC (by the network orderedness assumption), so $\texttt{U}_0$ was given its value at the CC before $\texttt{U}_1$ is given its value. Then, by definition of $\preceq_{\texttt{CC}}$, it must be the case that $\texttt{U}_0 \preceq_{\texttt{CC}} \texttt{U}_1$.

- Sub-Case B: U_1 is a local write. Since S processes $U_\emptyset$, the RRESP, before the local write U_1, the RREQ which triggered $U_\emptyset$ must have arrived to the CC before the WRITE sent by S to the CC in response to U_1. Therefore, the WRITE associated with U_1 must have arrived to the CC before the WRITE associated with $U_\emptyset$. Thus, by the definition of $\preceq_{CC}$, $U_\emptyset \preceq_{CC} U_1$.
- Sub-Case C: U_1 is a WRITE (received by S from the CC). Then, U_1 will have hit the CC after the RREQ associated with $U_\emptyset$. Otherwise, S would not yet have been a sharer of x and therefore would not have been sent U_1 (a WRITE) in the first place. Therefore, $U_\emptyset \preceq_{CC} U_1$.

Case 2: $U_\emptyset$ is a local write. We now case on U_1.

- Sub-Case A: U_1 is a RRESP. Then, x was evicted between updates $U_\emptyset$ and U_1; otherwise the read associated with U_1 would have read from a local cache (via a hit) instead of sending a RREQ to the CC. The WRITE associated with the local write $U_\emptyset$ sent by S will hit the CC before the RREQ associated with U_1 since messages hit the CC in the order they were sent (due to the ordered network). Therefore, U_1 will have an associated WRITE that is at least as new as $U_\emptyset$, so $U_\emptyset \preceq_{CC} U_1$.
- Sub-Case B: U_1 is a local write. Local WRITEs from S hit the CC in the order they were sent, and $U_\emptyset$ occurred at S before U_1, meaning its associated WRITE was sent to the CC before the WRITE associated with U_1. Therefore, the CC will order the WRITEs associated with $U_\emptyset$ and U_1 such that $U_\emptyset \prec_{CC} U_1$.
- Sub-Case C: U_1 is a WRITE (received by S from the CC). S is not registered as a sharer of x until $U_\emptyset$'s WRITE is processed by the CC by our base case assumption. Therefore, the WRITE associated with U_1 must have hit the CC after $U_\emptyset$, otherwise U_1 would not have been sent to S. Therefore, $U_\emptyset \preceq_{CC} U_1$.

Note that there is no case in which $U_\emptyset$ is a WRITE because $U_\emptyset$ is assumed to be the first update ever to shim[S][x]. This means that prior to $U_\emptyset$, S was not a sharer of x, so $U_\emptyset$ would not have been sent to S in the first place had it been a WRITE.

Induction Hypothesis. Suppose for the first $n > 0$ updates $U_\emptyset$...U_n to shim[S][x], $U_\emptyset \preceq_{CC} U_1 \preceq_{CC} \ldots \preceq_{CC} U_n$.

Induction Step. We want to show that for the next update U_{n+1} to shim[S][x], $U_n \preceq_{CC} U_{n+1}$. We first case on U_{n+1}.

Case 1: U_{n+1} is a RRESP. We now case on U_n.

- Sub-Case A: U_n is a RRESP. This is the same case as Case 1.A of the base case.
- Sub-Case B: U_n is a local write. This is the same as Case 2.A of the base case.
- Sub-Case C: U_n is a WRITE. Then, x was evicted from S between U_n and U_{n+1}; otherwise, the read which triggered U_{n+1} would have instead read from a local cache (via a hit). Since U_n arrived at S before the eviction, and the RREQ associated with U_{n+1} was sent to the CC after the eviction, then the WRITE associated with U_n must have hit the CC before the RREQ associated with (and thus, the WRITE associated with) U_{n+1}. Therefore, $U_n \preceq_{CC} U_{n+1}$.

Case 2: U_{n+1} is a local write. We now case on U_n.

- Sub-Case A: U_n is a RRESP. This case is the same as Case 1.B of the base case.
- Sub-Case B: U_n is a local write. This case is the same as Case 2.B of the base case.
- Sub-Case C: U_n is a WRITE (received by S from the CC). We know that the WRITE associated with U_{n+1} has not yet arrived at the CC at the time it is written to S, since local writes are written locally before their corresponding WRITE gets sent to the CC. Since U_n was already written to S's cache by the time U_{n+1} updates S's cache, then U_n's associated WRITE must have been processed at the CC before U_{n+1} performs at S, and therefore also before the WRITE associated with U_{n+1} is processed at the CC. Thus, $U_n \preceq_{CC} U_{n+1}$.

Case 3: U_{n+1} is a WRITE. We now case on U_n.

- Sub-Case A: U_n is a RRESP. This case is the same as Case 1.C of the base case.
- Sub-Case B: U_n is a local write. We will show that it is impossible for the WRITE associated with U_n to have arrived at the CC after the WRITE associated with U_{n+1}. Let k be equal to shim[S][x].wc_local after U_n has been processed at S, but suppose only l of these k local writes hit the CC before U_{n+1}. We will show that $l = k$, which would mean that $U_n \preceq_{CC} U_{n+1}$.

  Let m_n be equal to shim[S][x].wc_non_local after U_n has been processed at S, and let m_{n+1} be the number of non-local WRITEs to x that hit the CC and get forwarded to S between shim[S][x].W_{sync}, the most recent synchronizing write to x at S per Definition 1.4, and the WRITE associated with U_{n+1}. We can decompose U_{n+1}.ts as follows: U_{n+1}.ts = shim[S][x].$ts_{sync} + l + m_{n+1} + 1$. By Lemma (Timestamp Invariant), after U_n is performed, shim[S][x].ts = shim[S][x].$ts_{sync} + k + m_n$. By Lemma (Non-Local Writes Increment the Timestamp), and by our assumption that U_{n+1} updated S's cluster's LLC (and therefore passed the timestamp check), we know that shim[S][x].ts + 1 must be equal to U_{n+1}.ts. Plugging the decomposed values in, this gives us shim[S][x].$ts_{sync} + k + m_n + 1$ = shim[S][x].$ts_{sync} + l + m_{n+1} + 1$. Since all non-local writes that end up at S must have gone through the CC, and WRITEs from the CC arrive in the order they were sent due to our network orderedness assumption, we know that $m_n = m_{n+1}$. Simplifying the previous inequality, we get $l = k$, proving that $U_n \preceq_{CC} U_{n+1}$.

- Sub-Case C: $\mathtt{U_n}$ is a WRITE. WRITEs arrive in the order they were sent from the CC due to the orderedness of the network, so $\mathtt{U_n} \preceq_{\mathtt{CC}} \mathtt{U_{n+1}}$.

Therefore, by induction, we have proven that for any shim S, address x, and pair of updates $\mathtt{U_i}, \mathtt{U_j}$ to shim[S][x] such that $\mathtt{U_i}$ updates shim[S][x] before $\mathtt{U_j}$, $\mathtt{U_i} \preceq_{\mathtt{CC}} \mathtt{U_j}$.

Lemma (SC-Per-Location*): for all addresses a, $\exists \prec_{\mathtt{a}}$ a system-wide total order on all writes to a, such that for any pair of reads $\mathtt{R} \rightarrow_{\mathtt{sb}^*} \mathtt{R'}$ that read from writes w and w′ to a, respectively, $\mathtt{w} \preceq_{\mathtt{a}} \mathtt{w'}$.

Proof. We prove this claim by proving that $\prec_{\mathtt{CC}}$, the order in which WRITEs hit the CC, is our desired order $\prec_{\mathtt{a}}$.

Consider a shim S and address a. Assume for sake of contradiction that the order of some reads violates $\prec_{\mathtt{CC}}$, meaning there are two reads R0 and R1 of a at S, where $\mathtt{R0} \rightarrow_{\mathtt{sb}^*} \mathtt{R1}$, R0 reads from some write W0 and R1 reads from W1, but $\mathtt{W1} \prec_{\mathtt{CC}} \mathtt{W0}$. The values W0 and W1 returned to reads R0 and R1 will be the values most recently written to S's cache, either by WRITEs, local writes, or RRESPs. By Lemma (update order), all updates to S at address a happen in $\preceq_{CC}$ order, so $\mathtt{W0} \preceq_{\mathtt{CC}} \mathtt{W1}$. This contradiction proves our claim.

Lemma (rf-cc): If an rf edge exists between a read and a write, then the write must have hit the CC before the read either hit the CC, or completed locally.

Proof. Case on how the read reads its value.

- Case 1: From the CC. If a read reads a value from the CC, then a RRESP message is sent back to the shim with this data. Clearly, the data carried by the RRESP will have to have hit the CC before the RREQ message arrived, and the value that is eventually returned to the read is the value carried by the RRESP. So the claim holds.
- Case 2: From the shim cache. In order for the write to have arrived at the shim, it has to have gone through the CC and then been sent to the shim as a write update. Therefore, it hit the CC before the read completed locally.

In any case, our claim holds.

Lemma (sb-cc): For $I \rightarrow_{\mathtt{sb}} I'$ such that I and I' are writes, fences, or reads that retrieve their values via the CC, $I \prec_{CC} I'$.

Proof. Per MemGlue's design, messages arrive to the CC in the order they are sent from the shims. And messages from the shims are sent in sb order. Therefore, I will hit the CC before I'.

Lemma (hb-cc): $I \rightarrow_{\mathtt{hb}} I' \implies I \prec_{CC} I'$, for I and I' writes or fences.

Proof. By induction on the number of writes/fences in the hb chain after I.

BC: $I \rightarrow_{\mathtt{sb}} I'$. By Lemma (sb-cc), $I \prec_{CC} I'$.

BC: $I \rightarrow_{\mathtt{sw}} I'$. Then I' is a fence by the definition of sw. In order for I to be related to I' by sw, there must be a read R such that $R \rightarrow_{\mathtt{sb}} I'$, and $W' \rightarrow_{\mathtt{rf}} R$ for W' the write associated with I. By the definition of associated write, $I = W'$ or $I \rightarrow_{\mathtt{sb}} W'$. In the latter case, by Lemma (sb-cc), $I \prec_{CC} W'$. By Lemma (rf-cc), W' must be in the CC before R completes, and by the design of MemGlue, I' cannot be sent to the CC until R completes. Therefore, $I \prec_{CC} I'$.

IH: Suppose for an hb chain containing n writes/fences I_0, ..., I_n, such that $I \rightarrow_{\mathtt{hb}} I_0 \rightarrow_{\mathtt{hb}} \ldots \rightarrow_{\mathtt{hb}} I_n$, and $I \prec_{CC} I_n$.

IS: Consider the following structure: $I \rightarrow_{\mathtt{hb}} I_0 \rightarrow_{\mathtt{hb}} \ldots \rightarrow_{\mathtt{hb}} I_n \rightarrow_{\mathtt{hb}} I_{n+1}$. We want to show that $I \prec_{CC} I_{n+1}$.

If I_n is a write, then by the IH, $I \prec_{CC} I_n$. If I_n is a fence, then let W' be the write associated with I_n. By the IH $I \prec_{CC} I_n$, and by MemGlue's design, instructions are sent to the CC in program order, so $I_n \prec_{CC} W'$, so $I \prec_{CC} W'$.

We just established that $I \prec_{CC} W'$. Now, case on the specific hb edge type between I_n and I_{n+1}.

- Case 1: sb. Then by MemGlue's protocol design, writes and fences hit the CC in sb order, so $I_n \prec_{CC} I_{n+1}$. Therefore, $I \prec_{CC} I_{n+1}$.
- Case 2: sw. Then I_{n+1} must be a fence in order to be the sink of a sw edge. By Lemma (rf-cc), the read associated with I_{n+1} does not complete until after W' hits the CC, and I_{n+1} cannot be sent to the CC until after its associated read completes, by MemGlue's design. Therefore, $I \prec_{CC} W' \prec_{CC} I_{n+1}$, so $I \prec_{CC} I_{n+1}$.

Therefore, by induction, we have proven our lemma.

Lemma (sw-seen): for instructions $I1$ and $I2$, if $I1 \rightarrow_{\mathtt{sw}} I2$, then $I1 \rightarrow_{\mathtt{seen}} I2$.

Proof. Note that there are two pieces of the seen definition: 1) $\forall R$ such that R is a read, $I = R$ or $I \rightarrow_{\mathtt{sb}} R$, and addr($R$) = addr($W$), $W \preceq_{CC}$ val(R), and 2) $\forall WF'$ such that WF' is a write or fence, and $I = WF'$ or $I \rightarrow_{\mathtt{sb}} WF'$, or $I = WF'$, $W \prec_{CC} WF'$. Thus, for each case we will be proving two claims, which we will refer to as $\mathtt{seen}_1$ and $\mathtt{seen}_2$.

Consider the write associated with $I1$, call this W, and the read associated with $I2$, call this R. By Lemma (rf-cc), we know that W hit the CC before R either hit the CC or completed locally. Since $I1$ is either W or a fence that was sb-before W, then $I1$

hit the CC before R completed. And since $I2$ is either R or a fence that was sb-after R, then $I1$ hit the CC before $I2$ either hit the CC or completed locally. Now case on the seen definition.

$\underline{\mathsf{seen}_2}$: Consider some write or fence WF' such that $WF' \rightarrow_{\mathsf{sb}} I2$. Then WF' cannot hit the CC until after $I2$ completes locally, whether $I2$ is a read or fence. By our previous reasoning, this implies that WF' cannot hit the CC until after $I1$ has hit the CC, so $I1 \rightarrow_{\mathsf{seen}} WF'$. Thus, seen_2 is upheld.

$\underline{\mathsf{seen}_1}$: Consider some R' such that $I2 \rightarrow_{\mathsf{sb}} R'$, and addr($R'$) = addr($W$) (in this case, it must be that $I1$ is a write). Case on how R' reads its value.

- Case 1: From the CC. Then we know by the sb edge between $I2$ and R' that R' will not hit the CC until after $I2$ completes locally, meaning R' will hit the CC after W. Therefore, it must read a value at least as new as W, since the CC decides the order of writes and therefore only overwrites values with newer ones.
- Case 2: From the shim cache. We already know that R read from W due to the sw edge between them, and it would have updated the shim cache with W. By Lemma (SC-per-location), R' cannot then read a value older than W. So $W \rightarrow_{\mathsf{seen}} R'$.

Thus, in any case, if $I1 \rightarrow_{\mathsf{sw}} I2$, then $I1 \rightarrow_{\mathsf{seen}} I2$.

Lemma (hb-seen-sw): for write $W0$ and instructions $I1$ and $I2$, if $W0 \rightarrow_{\mathsf{hb}} I1 \rightarrow_{\mathsf{sw}} I2$, then $W0 \rightarrow_{\mathsf{seen}} I2$.

Proof. By Lemma (hb-cc), $W0 \prec_{CC} I1$. By Lemma (sb-cc), the write W associated with $I1$ is either $I1$ or hits the CC after $I1$, so $W0 \prec_{CC} W$. By Lemma (rf-cc), the read R associated with $I2$ hits the CC or completes locally after W hits the CC. Now, we again case on the two parts of the seen definition, as in Lemma (sw-seen).

$\underline{\mathsf{seen}_2}$: Consider some write or fence WF' such that WF' is $I2$ or $WF' \rightarrow_{\mathsf{sb}} I2$. WF' is sb-after R, meaning it cannot hit the CC until after R completes by MemGlue's design. Combining this with our previous $\prec_{CC}$ reasoning, we know that $W0 \prec_{CC} WF'$, so seen_2 holds.

$\underline{\mathsf{seen}_1}$: Consider some read R' such that $R \rightarrow_{\mathsf{sb}} R'$, and addr($R$) = addr($W0$). Case on the presence of addr($W0$) in R's shim at the time W performed at the CC.

- Case 1: addr($W0$) was not shared by R's shim, so no update was sent to R's shim. If it is still not shared by the time R' performs, then R' will have to go through the CC. However, since $W0$ had already hit the CC at this point, R' will read a value at least as new. Otherwise, if R's shim became a sharer after $W0$ hit the CC but before R' performed, then some previous instruction brought addr($W0$) into the shim cache. Again, since $W0$ already hit the CC when that instruction retrieved the most recent data, it would have brought a value at least as new as $W0$ into the shim cache. Therefore, in any case, seen_1 is upheld.
- Case 2: addr($W0$) was shared by R's shim when it hit the CC, so an update of W is on the way. Case on how R reads its value.
  - Case 1: through the shim cache. Then an update of W was sent and updated the shim cache. Realize that $W0$ hit the CC before W, so its update was sent to R's shim before W's update. By the orderedness of the network, then $W0$ updated the shim cache before W. Therefore, by Lemma (update order), R' will read a value at least as new as $W0$.
  - Case 2: From the CC. Then R' hit the CC after R completed locally, meaning it hit the CC after W hit the CC. This means that R' hit the CC after $W0$, meaning it would have read a value at least as new.

  In any case, R' reads a value at least as new as $W0$, so seen_1 is upheld.

Therefore, both parts of the seen definition are upheld, so $W0 \rightarrow_{\mathsf{seen}} I2$.

Lemma (rf-hb-cc): $W0 \rightarrow_{\mathsf{rf}} R0 \rightarrow_{\mathsf{hb}} I1 \implies W0 \prec_{CC} I1$, for $I1$ a write or fence.

Proof. By induction on the number of edges in the hb chain.

BC. hb = sb. By Lemma (rf-cc), $R0$ hits the CC, or performs locally, after $W0$ hit the CC. And, $I1$ cannot perform until $R0$ performs locally or hits the CC, so it arrives at the CC after $W0$.

BC. hb = sb; sw. This means for some instruction $I2$, $W0 \rightarrow_{\mathsf{rf}} R0 \rightarrow_{\mathsf{sb}} I2 \rightarrow_{\mathsf{sw}} I1$ (and $I1$ must be a fence in this case). By Lemma (rf-cc), we know that $R0$ hit the CC or performed locally after $W0$ hit the CC. We also know that $I2$ (a write or a fence, per the sw edge definition) cannot hit the CC until $R0$ has either performed locally or hit the CC since they are sb-related, so $I2$ hits the CC after $W0$. We also know that there is some read $R1$ sb-before $I1$ that reads from $I2$ or an sb-future write, per the definition of sw, and by Lemma (rf-cc) we know that $R1$ hits the CC or performs locally before the write that it reads from, meaning it hits the CC or performs locally before $I2$. And $I1$ cannot hit the CC until after $R1$ performs locally/hits the CC, so per transitivity, $I1$ must hit the CC after $W0$.

IH. Suppose for an hb chain starting with $R0$, and n writes/fences following, each write/fence hits the CC after $W0$.

IS. Suppose there is an hb-chain ending with $I1$, the $n+1^{\text{th}}$ write/fence. We want to show that $I1$ hits the CC after $W0$. We know that the n^{th} write/fence, I_n, hit the CC after $W0$. There are three cases for how $I1$ relates to I_n.

- Case 1: sb. Then we know I_n was processed at the CC before $I1$ because instructions are sent to the CC in sb order, so by transitivity, we know that $W0$ hit the CC before $I1$.
- Case 2: sw. Case on I_n.
  - I_n is a write. This means that I_n was read from by a read sb-before $I1$. By Lemma (rf-cc), we know that I_n hit the CC before the read that read from it performed, and $I1$ performed after this read, meaning $I1$ hit the CC after $W0$.
  - I_n is a fence. This means that I_n is sb-before a write W that gets read from by a read R that is sb-before $I1$. Then I_n hit the CC before W, and by Lemma (rf-cc), W hit the CC before R performed locally/hit the CC. $I1$ cannot hit the CC until after R performs locally/hits the CC, meaning by transitivity that $I1$ hit the CC after $W0$.
- Case 3: sw; sb. Then there is some read R' such that $I_n \rightarrow_{\mathsf{sw}} R' \rightarrow_{\mathsf{sb}} I1$. By Lemma (sb-cc) and Lemma (rf-cc), we know that I_n hit the CC before R' either hit the CC or completed locally, and R' must hit the CC or complete locally before $I1$ can be sent to the CC, per MemGlue's specification. Therefore, $I_n \prec_{CC} I1$, and by transitivity, $W0 \prec_{CC} I1$.

Therefore, by induction, we have shown that $W0 \prec_{CC} I1$.

Lemma (rf-hb-seen-sw): $W0 \rightarrow_{\mathsf{rf}} R0 \rightarrow_{\mathsf{hb}} I0 \rightarrow_{\mathsf{sw}} I1 \wedge W0 \rightarrow_{\mathsf{seen}} I0 \implies W0 \rightarrow_{\mathsf{seen}} I1$.

Proof. By Lemma (rf-hb-cc), $W0 \prec_{CC} I0$. And we know by Lemma (rf-cc) that the write W associated with $I0$ hit the CC before the read R associated with $I1$ hit the CC or completed locally. Since W is either $I0$ or is sb-after $I0$, then $W0 \prec_{CC} W$, and $W0$ hit the CC before R completed locally. Now consider the two parts of the seen definition, as in Lemma (sw-seen).

seen_2: Consider some write or fence WF' such that $WF' = I1$, or $I1 \rightarrow_{\mathsf{sb}} WF'$. Realize that $I0 \rightarrow_{\mathsf{hb}} WF'$ in either case, so by Lemma (hb-cc), $I0 \prec_{CC} WF_{S'}$. And as previously stated, we know that $W0 \prec_{CC} I0$. Combining this reasoning, we know that $W0 \prec_{CC} WF'$, proving seen_2.

seen_1: Consider some read R' such that $R' = I2$ or $I2 \rightarrow_{\mathsf{sb}} R'$. Case on how R' reads its value.

- Case 1: From the CC. Then R' must hit the CC after R completes locally or hits the CC, since it is after it in sb order. As previously shown, $W0$ hits the CC before R completes locally or hits the CC, so it will already be in the CC by the time R' hits the CC. Then R' must read a value at least as new as $W0$, showing $W0 \rightarrow_{\mathsf{seen}} I1$.
- Case 2: From the shim cache. In the event that addr($W0$) was not being shared by the shim where R' performed (call this S) at the time $W0$ hit the CC, then no write update of $W0$ is on the way. However, then some previous instruction that was sb-before R' brought addr($W0$) into S's cache. This instruction must have hit the CC after $W0$, otherwise an update of $W0$ would have been sent to S. Then, this instruction would have brought a value at least as new as $W0$ into the shim cache, and by Lemma (update order) R' would have read a value at least as new.

  Otherwise, suppose an update of $W0$ is on the way to S. Case on how R reads its value.
  - Case 1: From the CC. Then its RREQ would have hit the CC after $W0$ as previously established, so the update of $W0$ would have been sent to S before the RRESP. R cannot complete until the RRESP comes back, and by the orderedness of the network, the update of $W0$ will arrive before the RRESP and update the shim cache. Then, by Lemma (update order), we know that R' will read a value at least as new.
  - Case 2: From the shim cache. Then W arrived as a write update and updated the shim cache. Again, since $W0 \prec_{CC} W$, and messages do not get reordered in the network, then the write update of $W0$ was sent to and received by S before W. Therefore, $W0$ will have updated the shim cache before R' performs, and by Lemma (update order), R' will read a value at least as new.

  Therefore, in any case, seen_1 is upheld.

We have proven seen_1 and seen_2, so $W0 \rightarrow_{\mathsf{seen}} I1$.

Lemma (hb-seen): $W \rightarrow_{\mathsf{hb}} I \implies W \rightarrow_{\mathsf{seen}} I$.

Proof. By induction on the number of instructions in the hb chain. We again prove seen_1 and seen_2 separately as in Lemma(sw − seen).

BC: $W \rightarrow_{\mathsf{sb}} I$. If seen_1 is not upheld, then there is some read R at shim S such that $W \rightarrow_{\mathsf{sb}} R$, but val($R$) $\prec_{CC} W$. However, R either read its value from S's cache, or from a RRESP. So either the RRESP updated S's cache after servicing R, or a prior write update updated S's cache, to later be read from by R. But by Lemma (update order), this is not possible for an update to contradict $\prec_{CC}$ order at S, so seen_1 is upheld.

If seen_2 is not upheld, then there is some write or fence WF' such that $I = WF'$ or $I \rightarrow_{\mathsf{sb}} WF'$, but $WF' \prec_{CC} W$. This also violates Lemma (update order), so seen_2 is upheld.

BC: $W \rightarrow_{\mathsf{sw}} I$. Then by Lemma (sw-seen), $W \rightarrow_{\mathsf{seen}} I$.

IH: Suppose for an hb chain with $n \geq 2$ instructions, $W \rightarrow_{\mathsf{hb}} I_n \implies W \rightarrow_{\mathsf{seen}} I_n$.

IS: Consider an $(n+1)$-length hb chain. We want to show that $W \rightarrow_{\mathsf{hb}} I_{n+1} \implies W \rightarrow_{\mathsf{seen}} I_{n+1}$. Case on the final edge.

- Case 1: sb. By the IH, $W \rightarrow_{\mathsf{seen}} I_n$. So for any read R s.t. $I_n = R$ or $I_n \rightarrow_{\mathsf{sb}} R$, seen_1 holds, and for any write or fence WF' s.t. $I_n = WF'$ or $I_n \rightarrow_{\mathsf{sb}} WF'$, seen_2 holds. These universal quantifications mean that these properties hold for I_{n+1} as well.
- Case 2: by Lemma (hb-seen-sw), this case is proven.

Therefore, by induction, we have proven our claim.

Lemma (rf-hb-seen): $R \rightarrow_{\mathsf{hb}} I \implies val(R) \rightarrow_{\mathsf{seen}} I$

Proof. If R reads from initial memory, then we are trivially done. So suppose there is some write W such that $W \rightarrow_{\mathsf{rf}} R \rightarrow_{\mathsf{hb}} I$. We want to show that $W \rightarrow_{\mathsf{seen}} I$. We will again be proving seen_1 and seen_2. Let W be the shim on which R performs.

seen_2: By Lemma (rf-cc) we know that W is in the CC by the time R completes. Now case on the hb edge.

- Case 1: hb is only sb edges. Then $W \rightarrow_{\mathsf{rf}} R \rightarrow_{\mathsf{sb}} I$. By the definition of seen_2, we must show that for any write or fence WF' such that $I = WF'$ or $I \rightarrow_{\mathsf{sb}} WF'$, $W \prec_{CC} WF'$. Consider some such WF'. Then by Lemma (rf-cc), W is in the CC by the time R completes. And any instruction that is sb-after R cannot perform until R completes, meaning it won't be able to hit the CC until after W. Therefore, for this arbitrary WF' that is equal to or sb-after I, $W \prec_{CC} WF'$.
- Case 2: hb has some sw edge. Then $W \rightarrow_{\mathsf{rf}} R \rightarrow_{\mathsf{sb}} I' \rightarrow_{\mathsf{hb}} I$, where I' is the instruction on R's shim with an outgoing sw edge (so it must be a write or fence). By the reasoning in Case 1, $W \prec_{CC} I'$. And by Lemma (hb-cc), $I' \prec_{CC} I$. Therefore, $W \prec_{CC} I$. This means that for any write or fence WF' that is either equal to or sb-after I, since WF' cannot be sent to the CC until after I completes, $W \prec_{CC} WF'$.

Therefore, in either case, seen_2 holds.

seen_1: Consider some arbitrary read R' such that $I = R'$ or $I \rightarrow_{\mathsf{sb}} R'$. We want to show that $W \preceq_{CC} \text{val}(R')$. We induct on the number of instructions in the hb chain.

BC 1: $W \rightarrow_{\mathsf{rf}} R \rightarrow_{\mathsf{sb}} I$. Then there must be some update to S's cache updating addr(W) to W, either because R retrieved the value from the CC, or it arrived as a WRITE update. Then by Lemma (update order), R' cannot read a value older than W. So seen_1 holds.

BC 2: $W \rightarrow_{\mathsf{rf}} R \rightarrow_{\mathsf{sb}} I' \rightarrow_{\mathsf{sw}} I$. By BC 1 and Lemma (rf-hb-seen-sw), we know that $W \rightarrow_{\mathsf{seen}} I$.

IH: Suppose for an hb chain with $n \geq 2$ instructions, $W \rightarrow_{\mathsf{rf}} R \rightarrow_{\mathsf{hb}} I_n \implies W \rightarrow_{\mathsf{seen}} I_n$.

IS: Consider an $(n+1)$-length hb chain. We want to show that $W \rightarrow_{\mathsf{rf}} R \rightarrow_{\mathsf{hb}} I_{n+1} \implies W \rightarrow_{\mathsf{seen}} I_{n+1}$. Case on the final edge.

- Case 1: sb. By the IH, $W \rightarrow_{\mathsf{seen}} I_n$. So $\forall R$ s.t. R is a read, $I_n = R$ or $I_n \rightarrow_{\mathsf{sb}} R$, seen_1 holds. This universal quantification means that seen_1 holds for I_{n+1} as well.
- Case 2: by Lemma (rf-hb-seen-sw), this case is proven.

Therefore, by induction, we have proven our claim.

1.1.2 Coherence Axiom.

Proof goal: if hb; eco is reflexive in some execution of a program, then MemGlue disallows the execution.

Proof. Consider some reflexive hb; eco edge involving $I1$ and $I2$: $I1 \rightarrow_{\mathsf{hb}} I2 \rightarrow_{\mathsf{eco}} I1$. Case on the eco edge:

- rf
  We have the following structure: $W \rightarrow_{\mathsf{rf}} R \rightarrow_{\mathsf{hb}} W$. By Lemma (rf-hb-cc), $W \prec_{CC} W$. This is clearly a contradiction because an instruction cannot hit the CC before itself.
- mo
  We have the following structure: $W1 \rightarrow_{\mathsf{mo}} W2 \rightarrow_{\mathsf{hb}} W1$. By Lemma (hb-cc), $W2 \prec_{CC} W1$. However, by mo order, which is decided by the CC, $W1 \prec_{CC} W2$, a contradiction.
- fr
  So we have the following structure: $R \rightarrow_{\mathsf{fr}} W \rightarrow_{\mathsf{hb}} R$. By Lemma (hb-seen), we know that $W \rightarrow_{\mathsf{seen}} R$. However, this would mean by the definition of seen that $W \prec_{CC} \text{val}(R)$. However, this contradicts the fr edge between R and W.
- fr; rf
  We have the following structure: $R1 \rightarrow_{\mathsf{hb}} R0 \rightarrow_{\mathsf{fr}} W1 \rightarrow_{\mathsf{rf}} R1$. By Lemma (rf-hb-seen), $W1 \rightarrow_{\mathsf{seen}} R0$. However, this means that $val(R0) \prec_{CC} W1$, which contradicts the fr edge between $R0$ and $W1$.
- mo; rf
  We have the following structure: $W1 \rightarrow_{\mathsf{rf}} R1 \rightarrow_{\mathsf{hb}} W0 \rightarrow_{\mathsf{mo}} W1$. By Lemma (rf-hb-cc), $W0 \prec_{CC} W1$. However, this contradicts mo order, which implies that $W0 \prec_{CC} W1$. So we have a contradiction.

1.1.3 Atomicity Axiom.

The hypothetical extensions in MemGlue to support RMW instructions would be the following:

- One link register per shim, stored at the CC
- On LL sent to CC, set the source shim's link register to that address and perform the read (make the LL shim a sharer)
- On a store, check each link register to see if that address is present. If so, clear it
- On SC sent to CC, if the link register is set, perform the store, otherwise do not and send back a failure acknowledgement

Proof Goal (Atomicity): if RMW intersected with (fr; mo) is non empty, or RMW intersected with eco is non empty for some execution, then the execution should not be observable in MemGlue.

Proof. We are trying to show that a LL-SC instruction pair cannot also be related by fr; mo. This means that if an LL instruction is related to its SC instruction by fr; mo, then the SC must not commit at the CC.

Suppose we have a LL/SC pair in MemGlue. Then in order for the LL to be fr-related to a write, the write must hit the CC after the LL, otherwise the LL would have read from that write. The LL will set the link register to x and the valid bit to 1, and the write of x will reset the valid bit to 0.

Then the SC hits the CC after the write, because they are related by mo. The link register will be cleared by then, so the SC will fail, and the RMW edge will not be present in the litmus test. In order for the LL/SC pair to perform and for the RMW edge to be added, the LL will need to be retried. But, when the LL is performed again, the LL and write of x will not be related by fr since the write is already in the CC by the time the second LL is performed. Then the litmus test would not have this fr; mo edge. Therefore, this behavior is not observable in MemGlue.

1.1.4 SC Axiom.

Edge Descriptions

- scb = hb|mo|fr
- psc_base = I1; scb; I2
  - I1 can be either some SC instruction, or $F \rightarrow_{\mathsf{hb}} I$ for any instruction I.
  - I2 can be either some SC instruction, or $I \rightarrow_{\mathsf{hb}} F$ for any instruction I.
- psc_fence = F; (hb|(hb; eco; hb)); F
- psc = psc_base|psc_fence

Note that we make a small deviation from the scb edge description in (cite RC11). RC11 weakened the scb relation because for Itanium processors, due to some intricacies with fences, it is not true that scb = (hb | mo | fr): if this were the case, there would be some programs that would be observable on Itanium processors, but would be fordibben by C11. Therefore, they weakened the scb edge, and thus the SC axiom. However, MemGlue treats scb as (hb | mo | fr), so we prove the SC axiom assuming scb = (hb | mo | fr). This means that MemGlue is stronger than RC11, which does not affect correctness.

Preliminary Claims

Lemma (hb-fr): if $I_{\mathrm{n}} \rightarrow_{\mathsf{hb}} R \rightarrow_{\mathsf{fr}} W$, for I_{n} a fence or SC write, then $I_{\mathrm{n}} \prec_{CC} W$.

Proof. Suppose R takes place at some shim S. Assume for sake of contradiction $W \prec_{CC} I_n$. Case on how R reads its value.

- Case 1: R reads from the CC. Case on the hb edge between I_n and R.
  - Case 1: hb = sb. Then since instructions hit the CC in sb order by the specification of MemGlue, the RREQ of R hits the CC after I_n, and therefore after W by our assumption. However, since the CC updates its cache lines in mo order, this means that R would have been returned a value at least as new as W in its RRESP. This contradicts the fr edge between R and W.
  - Case 2: hb contains after least one sw edge. Then there are some I', I'' such that $I_n \rightarrow_{\mathsf{hb}} I' \rightarrow_{\mathsf{sw}} R$, or $I_n \rightarrow_{\mathsf{hb}} I' \rightarrow_{\mathsf{sw}} I'' \rightarrow_{\mathsf{sb}} R$. In the former case, we know by Lemma (hb-cc) that $I_n \prec_{CC} I'$, and by Lemma (rf-cc) that R's RREQ hit the CC after the write associated with I', meaning it hit the CC after I' by Lemma (sb-cc). Therefore, R hit the CC after W, meaning it will read a value W or newer, contradicting the fr edge. In the latter case, we know by Lemma (hb-cc) that $I_n \prec_{CC} I'$, and by Lemma (rf-cc) that the read associated with I'' completed after the write associated

with I' hit the CC. Therefore, combining with Lemma (sb-cc) we know that I' hit the CC before I'' completed locally, and since instructions are processed in sb order we know that R's RREQ didn't hit the CC until after I'' completed locally. Therefore, W must be in the CC before R's RREQ arrives, meaning R will read a value at least as new as W. This contradicts the fr edge between them.

- Case 2: Suppose R reads from the shim cache. Note that we can assume that a write update of W was sent to R's shim – otherwise, it would be the case that addr(W) was not in R's shim cache by the time W happened. However, this would mean that another instruction on R's core brought addr(W) into the shim after W hit the CC, meaning the value brought into R's cache would be at least as new as W. This would contradict the fr edge between R and W. Therefore, we will assume that when W hit the CC, R's shim was a sharer and was sent a write update of W. Now case on the hb edge between I_n and R.
  - Case 1: hb = sb. Suppose I_n and R happen on some shim S. We know that a write update of W will be on the way to S before I_n reaches the CC, since it hit the CC before I_n by our assumption. Therefore, when I_n performs at S, since it is a W_{SC} or a fence, no other instruction at S may perform until it receives its acknowledgment back. The W update, having been sent before the acknowledgment, will already have arrived at S by the time I_n completes, and therefore that R that is sb-after I_n will eventually read a value at least as new as W (by Lemma (update order)). This would violate the fr edge between R and W.
  - Case 2: hb contains at least one sw, meaning we have some instructions I' and I'' such that $I_n \rightarrow_{hb} I' \rightarrow_{sw} R$ or $I_n \rightarrow_{hb} I' \rightarrow_{sw} I'' \rightarrow_{sb} R$. Case on the existence of a fence between I_n and R.
    - Case 1: There is some fence $F1$ along this hb between I_n and R (meaning $F1$ may be I_n, I', or I'').
      If $F1$ is I'' or any instruction on S, then R cannot perform before I'' completes per MemGlue's specification. And, W hit the CC before the FREQ of I'', and messages arrive in order, W's update arrived at S before I'' received back its FRESP and therefore could complete. R cannot perform until I'' completes per MemGlue's specification, so W will already be in S's shim cache by the time R performs. By Lemma (update order), R must read a value at least as new as W, contradicting the fr edge between them.
      So suppose $F1$ performs at some shim other than S. If $F1$ is I_n, then by assumption $W \prec_{CC} F1$, otherwise by Lemma (hb-cc), $I_n \prec_{CC} F1$, meaning $W \prec_{CC} F1$. So in any case, $W \prec_{CC} F1$. When $F1$ reaches the CC, it will send out FREQ's to every shim. Since the W update arrived at the CC before the FREQ, it was sent to S before the return FREQ. Also, by Lemma (hb-cc), $F1 \preceq_{CC} I'$ (since $F1$ may be I', it is $\preceq_{CC}$ instead of $\prec_{CC}$). We also know that for the write associated with I', which we call W', $I' \preceq_{CC} W'$ by the definition of associated write, so $F1 \prec_{CC} W'$ (here, we use $\prec_{CC}$ instead of $\preceq_{CC}$ because $F1$ is a fence, so if $F1 = I'$, then it must be that $I' \prec_{CC} W'$, and if $F1$ is not I', then $F1 \prec_{CC} I' \preceq_{CC} W'$, meaning $F1 \prec_{CC} I'$). Case on how R'', the read associated with I'', reads its value.
      - Case 1: From the shim cache, via a write update of W'. The write update of W' is sent from the CC after the FREQ of $F1$, since $F1 \prec_{CC} W'$.
      - Case 2: From the CC. Then the RRESP of R'' is ordered after W' at the CC because it read the value of W', meaning it must also be ordered after the FREQ of $F1$.

      In either case, this WRITE or W' or RRESP of R'' will arrive at S after $F1$'s FREQ, which must arrive after the update of W. Whether I' synchronizes with I'' or directly with R, W will be in S's shim cache before R can perform, meaning (by Lemma (update order)) that R will read a value at least as new as W. This contradicts the fr edge between R and W.
    - Case 2: the hb chain between I_n and R only contains writes and reads. Then we have some $W0$ and $R0$ such that: $I_n \rightarrow_{hb} W0 \rightarrow_{sw} R0$, and R is either $R0$ or is sb-after $R0$. Case on how $R0$ reads its value.
      - Case 1: $R0$ reads from the CC. Then $R0 \neq R$, since we are in the case that R reads from the shim cache. We know by assumption that $W \prec_{CC} I_n$, and by Lemma (hb-cc) that $I_n \prec_{CC} W0$. Since $R0$ reads from $W0$, it will be ordered after it at the CC, so it will also be ordered after the W at the CC. Since messages arrive in order, $R0$ will not receive its RRESP until after W arrives and updates the shim cache, meaning by Lemma (update order) that R will read a value at least as new as W. This contradicts the fr edge between them.
      - Case 2: $R0$ reads from the shim cache. Then R can either be $R0$ or sb-after $R0$. We know that $W0$ hit R's shim cache before $R0$ performs, otherwise $R0$ would not have read from $W0$. Since $I_n \rightarrow_{hb} W0$, then by Lemma (hb-cc), $I_n \prec_{CC} W0$, meaning W would have also hit the CC before $W0$. Therefore, we know that $W0$ arrives at S after W arrives and is written to the shim cache, since messages arrive in order. Therefore, R (whether or not it is equal to $R0$) will read from a value at least as new as W. This contradicts the fr edge between them.

Therefore, in any case, we have proven our claim.

Lemma (psc-cc): $I_1 \rightarrow_{\text{psc}^*} I_n \implies I_i \prec_{CC} I_j$, for psc$*$ a chain of psc edges, $1 \leq i < j \leq n$ and $I_1 ... I_n$ writes or fences.

Proof. By induction on the number of writes and fences in the chain of psc edges.

BC 1: $I1$ and $I2$ are the only writes/fences, and psc$*$ is a single psc edge.

- Case 1: psc is psc_base. Case on the psc_base edge type.
  - mo. By the definition of mo, mo-related instructions hit the CC in mo order, so $I1 \prec_{CC} I2$.
  - hb. By Lemma (hb-cc), $I1 \prec_{CC} I2$.
  - fr. fr relates a read and write, and since $I1$ and $I2$ are writes or fences, a single fr edge between $I1$ and $I2$ is not possible.
- psc_fence. psc_fence can either be an hb edge or a hb; eco; hb edge. If psc_fence is an hb edge, then by Lemma (hb-cc), $I1 \prec_{CC} I2$. So consider psc = hb; eco; hb. Case on the eco edge.
  - rf. Then we have $I1 \rightarrow_{\text{hb}} W \rightarrow_{\text{rf}} R \rightarrow_{\text{hb}} I2$. By Lemma (hb-cc), $I1 \prec_{CC} W$, and by Lemma (rf-hb-cc), $W \prec_{CC} I2$, so $I1 \prec_{CC} I2$.
  - mo. Then we have $I1 \rightarrow_{\text{hb}} W0 \rightarrow_{\text{mo}} W1 \rightarrow_{\text{hb}} I2$. By the definition of mo, $W0 \prec_{CC} W1$, and by Lemma (hb-cc), $I1 \prec_{CC} W0$ and $W1 \prec_{CC} I2$. So $I1 \prec_{CC} I2$.
  - fr. Then we have $I1 \rightarrow_{\text{hb}} R \rightarrow_{\text{fr}} W \rightarrow_{\text{hb}} I2$. By Lemma (hb-fr), $I1 \prec_{CC} W$, and by Lemma (hb-cc), $W \prec_{CC} I2$. So $I1 \prec_{CC} I2$.
  - mo; rf. Then we have $I1 \rightarrow_{\text{hb}} W0 \rightarrow_{\text{mo}} W1 \rightarrow_{\text{rf}} R \rightarrow_{\text{hb}} I2$. By Lemma (hb-cc), the definition of mo, Lemma (rf-hb-cc), and the usual transitivity of $\prec_{CC}$, $I1 \prec_{CC} I2$.
  - fr; rf. Then we have $I1 \rightarrow_{\text{hb}} R0 \rightarrow_{\text{fr}} W \rightarrow_{\text{rf}} R1 \rightarrow_{\text{hb}} I2$. By Lemma (hb-fr), Lemma (rf-hb-cc), and transitivity of $\prec_{CC}$, $I1 \prec_{CC} I2$.

BC 2: $I1$ and $I2$ are the only writes or fence, but psc$*$ consists of a chain of intermediate reads. This means we have: $I1 \rightarrow_{\text{psc}} R_1 \rightarrow_{\text{psc}} ... \rightarrow_{\text{psc}} R_k \rightarrow_{\text{psc}} I2$. The only psc edge that relates two reads is hb, so this simplifies to: $I1 \rightarrow_{\text{psc}} R_1 \rightarrow_{\text{hb}} R_k \rightarrow_{\text{psc}} I2$. The first psc can only be hb (this is the only psc edge type relating writes or fences to reads), so we have: $I1 \rightarrow_{\text{hb}} R_k \rightarrow_{\text{psc}} I2$. Case on the remaining psc edge.

- Case 1: hb. Then our chain simplifies to $I1 \rightarrow_{\text{hb}} I2$, so by Lemma (hb-cc) we know that $I1 \prec_{CC} I2$.
- Case 2: fr. Then our chain is $I1 \rightarrow_{\text{hb}} R_k \rightarrow_{\text{fr}} I2$. By Lemma (hb-fr), we know that $I1 \prec_{CC} I2$.

In any case, we have proven our claim in the base cases.

IH: Suppose for a psc chain involving n write or fence instructions $I_1 \rightarrow_{\text{psc}} ... \rightarrow_{\text{psc}} I_n$, for any $1 \leq i < j \leq n$ $I_i \prec_{CC} I_j$.

IS: Consider a chain with $n+1$ write or fence instructions $I_1 \rightarrow_{\text{psc}} ... \rightarrow_{\text{psc}} I_n \rightarrow_{\text{psc}} I_{n+1}$. By the IH, we know all $I_1 \ldots I_n$ writes and fences hit the CC in psc order, so we need only show that $I_n \prec_{CC} I_{n+1}$. Case on how I_n and I_{n+1} are related.

- Case 1: I_n and I_{n+1} are related by a single psc edge. We have already proven this in BC 1.
- Case 2: psc$*$ is made up only of intermediate reads, i.e. $I_n \rightarrow_{\text{psc}} R_1 \rightarrow_{\text{psc}} ... \rightarrow_{\text{psc}} R_k \rightarrow_{\text{psc}} I_{n+1}$. We have also already proven this in BC 2.

Therefore, by induction, we have proven our claim.

Lemma (psc-reads): A psc cycle must have at least one write or fence.

Proof. A psc cycle consists of psc_base and psc_fence edges. psc_fence edges always occur between two fences, so a fence-free psc cycle will only have psc_base edges. psc_base edges consist of scb edges between two SC instructions. scb edges are either hb, mo, or fr edges. However, only hb edges can occur between two read instructions. So if we have a psc cycle containing only reads, then we know that $R_0 \rightarrow_{\text{hb}} ... \rightarrow_{\text{hb}} R_n \rightarrow_{\text{hb}} R_0$. Now case on the hb edges' types.

- Case 1: Every hb edge in this cycle is made up only of sb edges. Then we would have an sb cycle, which is not possible because MemGlue executes all instructions in sb order.
- Case 2: There is at least one sw edge in the hb cycle, i.e. between two reads in the psc cycle R_i and R_j, we have that $R_i \rightarrow_{\text{hb}} I1 \rightarrow_{\text{sw}} I2$, and either $I2 = R_j$ or $I2 \rightarrow_{\text{hb}} R_j$. Consider the write associated with $I1$, call this $W1$, and the read associated with $I2$, call this $R2$. Then $W1 \rightarrow_{\text{rf}} R2$. However, $R2$ is either equal to $I2$, or $R2 \rightarrow_{\text{sb}} I2$, and $I2 \rightarrow_{\text{hb}} I1$ per our psc cycle, where $I1$ is either equal to $W1$ or $I1 \rightarrow_{\text{sb}} W1$. Therefore, per the definition of hb we have that $R2 \rightarrow_{\text{hb}} W1$, and $W1 \rightarrow_{\text{rf}} R2$. Realize that this outcome violates the coherence axiom, and we have already proven that MemGlue does not allow program outcomes that violate coherence, so this outcome is not observable in MemGlue.

Therefore, a psc cycle that only contains reads is not observable in MemGlue, so a psc cycle must contain at least one write or fence.

Proof Goal (SC): If a program execution exhibits a cycle of psc edges between instructions, the execution is not observable in MemGlue$_O$.

Proof. Consider an arbitrary psc cycle. By Lemma (psc-reads), this cycle must contain at least one write or fence. Let $I1$ be the write or fence. Then by Lemma (psc-cc), all writes and fences in a psc chain hit the CC in psc order. However, since the chain is cyclic, this means that $I1$ would have to have hit the CC before itself, which is a contradiction.

1.1.5 No-Thin-Air Axiom.

Lemma (sb-rf-cc): Suppose $R0$ is related to $W1$ by (sb | rf), meaning a chain of sb and rf relations, and $R0$ reads from some write $W0$. Then $W0 \prec_{CC} W1$.

Proof. By induction.

BC: $W0 \rightarrow_{\texttt{rf}} R0 \rightarrow_{\texttt{sb}} W1$. By the RF Edges claim, we know that $W0$ hit the CC before $R0$ either hit the CC or completed locally. And $W1$ cannot hit the CC until after $R0$ performs because they are related by sb. So $W0 \prec_{CC} W1$.

IH: Suppose for a (sb|rf) chain starting with $R0$ that contains n writes, $W0 \prec_{CC} W_i$ for all $0 < i \leq n$.

IS: Consider a (sb|rf) chain starting with $R0$, which contains $n+1$ writes. By the IH, we know the n^{th} write hit the CC after $W0$. Case on the relation between W_n and W_{n+1}.

- Case 1: sb. Then W_n and W_{n+1} are from the same shim, and messages hit the CC in sb-order. Therefore, W_{n+1} hits the CC after W_n, and thus after $W0$.
- Case 2: rf; sb. Then W_n gets read from by some read R_n, which happens sb-before W_{n+1}. By Lemma (rf-cc), R_n either hits the CC or completes locally after W_n hits the CC. And W_{n+1} cannot hit the CC until after R_n has completed. Therefore, $W_n \prec_{CC} W_{n+1}$.

Therefore, $W0 \prec_{CC} W_{n+1}$.

Proof Goal (No-Thin-Air): if (sb | rf) is cyclic in some execution of a program, then the execution is not observable in MemGlue.

Pf (No-Thin-Air). Suppose (sb|rf) is cyclic. Then there must be at least one rf edge, because sb by definition cannot be cyclic. Then we know we have the following structure: $W0 \rightarrow_{\texttt{rf}} R0 \rightarrow_{(\texttt{sb}|\texttt{rf})+} W0$.

By Lemma (sb-rf-cc), we know that $W0$ will have to hit the CC before itself. This is not possible, meaning that if (sb|rf) is cyclic in a litmus test, then MemGlue will not let that test be observable.

1.2 Proof of Correctness: Unordered MemGlue

Proof Goal: for each C11 axiom, if the axiom forbids an execution of a program, then this execution is not observable in MemGlue.

1.2.1 Preliminaries.

Notation. val(R) is the write W such that $W \rightarrow_{\texttt{rf}} R$.
Notation. addr(I) is the address associated with write/read I.
Definition. One instruction $I2$ has *seen* another instruction $I1$ ($I1 \rightarrow_{\texttt{seen}} I2$) if:

1. $\forall R$ such that R is a read, $I2 \rightarrow_{\texttt{sb}} R$ or $I2 = R$, and addr(R) = addr($I1$), then $I1 \preceq_{CC}$ val(R).
2. $\forall WF$ such that W is a write or fence, and $I2 \rightarrow_{\texttt{sb}} WF$ or $I2 = WF$, then $I1 \prec_{CC} WF$.

Definition. eco = rf | mo | fr | mo ; rf | fr ; rf
Definition: $\prec_{CC}$ denotes the order in which writes hit the CC. Note that we overload this definition when discussing fences: $\prec_{CC}$ also denotes the order in which writes *and* fences hit the CC.
Definition: The *write associated with a fence I* which has an outgoing sw edge is either I itself, if I is a write, or the write W such that $I \rightarrow_{\texttt{sb}} W$, and W being read from induced the outgoing sw edge on I.
Definition: The *read associated with a fence I* which has an incoming sw edge is either I itself, if I is a read, or the read R such that $R \rightarrow_{\texttt{sb}} I$, and R reading from some write induces the incoming sw edge on I.
Notation shim[S] denotes the shim S.

- shim[S][x] denotes the cache line associated with address x at shim S.
  - shim[S][x].data is the data stored for address x in the cluster associated with shim S.
  - shim[S][x].wc_non_local denotes the number of write updates to x *received* (whether or not they are *accepted*) from any shim S′ ≠ S since S became a sharer of x.
  - shim[S][x].wc denotes the write count on address x at shim S, meaning the number of writes to address x performed at the cluster associated with S since address x first became shared by S.
  - shim[S][x].$\texttt{ts}_{\texttt{sync}}$ denotes the first timestamp brought into shim[S][x] when that cache line goes from state I to V, meaning when it goes from unshared to shared by S. Per the protocol, a cache line becomes shared on a read or write miss, meaning $\texttt{ts}_{\texttt{sync}}$ will either be:
    1. the timestamp assigned to some write W on x, where W is read from by a read R that misses at shim S, or
    2. the timestamp assigned to the last write to x performed at the CC before S synchronized with the CC.
  - $\texttt{W}_{\texttt{sync}}$ will be the write whose timestamp became shim[S][x].$\texttt{ts}_{\texttt{sync}}$.
- shim[S].fenceCount denotes the number of FREQs received from the CC by shim S.

1.2.2 Proofs.

Lemma (fence order): if some message $M2$ is sent after a FREQ from the CC to some shim S, then M must accept before the FREQ at S.

Pf. Suppose an FREQ is sent from shim S_1 to the CC, then after it is processed, an FREQ will be sent to each shim. Suppose some message $M2$ was sent from the CC to some shim S_2 after the FREQ. Per MemGlue's specification, for each shim S, the CC counts the number of FREQs sent to S under CC.fenceCounters[S]. Therefore, if the FREQ sent to S_2 is the n^{th} FREQ to go to S_2, then M.fenceCount will be $\geq n$.

Now suppose $M2$ arrives at S_2 before the FREQ. Per MemGlue's specification, each shim also tracks a counter for the FREQs it has received. We know that shim[S_2].fenceCount will be less than n, since S_2 hasn't seen the n^{th} FREQ yet, and it only accepts FREQs in order per MemGlue's specification. Then $M2$.fenceCount > shim[S_2].fenceCount, meaning that $M2$ will not be accepted at S_2 (and cannot be accepted until at least the n^{th} fence has been accepted).

Lemma (timestamp invariant): for any shim S and address x, at any given time shim[S][x].ts = shim[S][x].$\texttt{ts}_{\texttt{sync}}$ + shim[S][x].wc + shim[S][x].wc_non_local.

Pf. By induction on the number of local and non-local writes performed at S. Suppose that S synced at address x on some write with timestamp shim[S][x].$\texttt{ts}_{\texttt{sync}}$.

BC: No local or non-local writes have performed, so shim[S][x].wc and shim[S][x].wc_non_local are both 0. Then S must have become a sharer of x via a local read, which would have set shim[S][x].ts to be $\texttt{shim[S][x].ts}_{\texttt{sync}}$. This is equal to $\texttt{shim[S][x].ts}_{\texttt{sync}}$ + shim[S][x].wc + shim[S][x].wc_non_local.
BC: $n \geq 1$ local writes have performed, and no non-local writes. Case on how S became a sharer or x.

- Case 1: on the first of these local writes. Then the first local write will synchronize with the CC, meaning a WRITE_ACK will be returned to S with the timestamp of the last write to x at the CC before S became a sharer (which by definition is $\texttt{shim[S][x].ts}_{\texttt{sync}}$), plus one. Then, the timestamp put into the shim cache will be $\texttt{shim[S][x].ts}_{\texttt{sync}}$ + shim[S][x].wc - 1 + 1, where shim[S][x].wc will be the number of local writes performed between the first local write and the return of the WRITE_ACK. This is equal to $\texttt{shim[S][x].ts}_{\texttt{sync}}$ + shim[S][x].wc + shim[S][x].wc_non_local.
- Case 2: on a previous local read. Then this read by definition set shim[S][x].ts to be $\texttt{shim[S][x].ts}_{\texttt{sync}}$. Then, each local write increments both shim[S][x].ts and shim[S][x].wc by 1, per MemGlue's specification, so after each local write, shim[S][x].ts = $\texttt{shim[S][x].ts}_{\texttt{sync}}$ + shim[S][x].wc, which is equal to $\texttt{shim[S][x].ts}_{\texttt{sync}}$ + shim[S][x].wc + shim[S][x].wc_non_local.

BC: 1 non-local write has accepted, no local writes, so shim[S][x].wc_non_local = 1, and shim[S][x].wc = 0. Then before the non-local write accepted, shim[S][x].ts was equal to $\texttt{shim[S][x].ts}_{\texttt{sync}}$. In order for the non-local write to have accepted, per MemGlue's specification its timestamp was shim[S][x].ts + msg.wCntr - shim[S][x].wc + 1, where msg.wCntr is the number of write updates to x sent from S that have been processed by the CC. Based on our assumption that only one local write and no non-local writes have performed, we know that before the non-local write performs, $\texttt{shim[S][x].ts} = \texttt{shim[S][x].ts}_{\texttt{sync}}$, shim[S][x].wc = 0, and msg.wCntr = 0. Therefore, the acceptance condition was $\texttt{msg.ts} = \texttt{shim[S][x].ts}_{\texttt{sync}} + 1$. When the non-local write then updates that shim cache, it will write its timestamp, $\texttt{shim[S][x].ts}_{\texttt{sync}} + 1$, which is equal to $\texttt{shim[S][x].ts}_{\texttt{sync}}$ + shim[S][x].wc + shim[S][x].wc_non_local.
IH: Suppose after each of $n \geq 0$ local and non-local writes, shim[S][x].ts is equal to $\texttt{shim[S][x].ts}_{\texttt{sync}}$ + shim[S][x].wc + shim[S][x].wc_non_local.
IS: Suppose an $n + 1^{\text{th}}$ write W_{curr} is performed at S.

- Case 1: W_{curr} is a local write. Then it increments both shim[S][x].ts and shim[S][x].wc, so we want to show that shim[S][x].ts + 1 = $\texttt{shim[S][x].ts}_{\texttt{sync}}$ + shim[S][x].wc + 1 + shim[S][x].wc_non_local. By the IH we know that shim[S][x].ts = $\texttt{shim[S][x].ts}_{\texttt{sync}}$ + shim[S][x].wc + shim[S][x].wc_non_local, so we can simplify our above equality to 1 = 1, which clearly holds.
- Case 2: W_{curr} is a non-local write. In order for it to have been accepted at the shim, by MemGlue's specification it would have been the case the msg.ts = shim[S][x].ts + msg.wCntr - shim[S][x].wc + 1. By the IH, we can swap out the shim[S][x].ts to result in the following equality:
  msg.ts = $\texttt{shim[S][x].ts}_{\texttt{sync}}$ + shim[S][x].wc + shim[S][x].wc_non_local + msg.wCntr - shim[S][x].wc + 1. This simplifies to the following:
  msg.ts = $\texttt{shim[S][x].ts}_{\texttt{sync}}$ + shim[S][x].wc_non_local + msg.wCntr + 1. Now case on if msg writes its data to the shim cache:
  - Case 1: msg writes its data. We will now argue that msg increments shim[S][x].ts.
    At the CC, msg's timestamp can be broken down into: msg.ts = CC.rCntr + CC.lCntr + $\texttt{shim[S][x].ts}_{\texttt{sync}}$, where CC.rCntr signifies the number of writes not from S that have updated x, and CC.lCntr signifies the number of writes originating from S that have updated x. Plugging this into S's acceptance logic, we know that:
    CC.rCntr + CC.lCntr + $\texttt{shim[S][x].ts}_{\texttt{sync}}$ = shim[S][x].ts + msg.wCntr - shim[S][x].wc + 1. Note that msg.wCntr is exactly CC.lCntr, so this simplifies to:
    CC.rCntr + $\texttt{shim[S][x].ts}_{\texttt{sync}}$ = shim[S][x].ts + - shim[S][x].wc + 1.
    Plugging in our induction hypothesis for shim[S][x].ts, we know that:
    CC.rCntr + $\texttt{shim[S][x].ts}_{\texttt{sync}}$ = $\texttt{shim[S][x].ts}_{\texttt{sync}}$ + shim[S][x].wc shim[S][x].wc_non_local - shim[S][x].wc + 1.
    This simplifies to: CC.rCntr = shim[S][x].wc_non_local + 1.
    So msg.ts = $\texttt{shim[S][x].ts}_{\texttt{sync}}$ + shim[S][x].wc_non_local + 1 + CC.lCntr. Now, it remains to show that CC.lCntr = shim[S][x].wc.
    In order for msg to have written its data, msg.ts > shim[S][x].ts. Plugging in our derived msg.ts value and our induction hypothesis for shim[S][x].ts, we know that: CC.lCntr + shim[S][x].wc_non_local + 1 + $\texttt{shim[S][x].ts}_{\texttt{sync}}$ > $\texttt{shim[S][x].ts}_{\texttt{sync}}$ + shim[S][x].wc + shim[S][x].wc_non_local. This simplifies to:
    CC.lCntr + 1 > shim[S][x].wc. This means that CC.lCntr ≥ shim[S][x].wc. However, by our assumption, more writes have occurred locally at S than have hit the CC, so CC.lCntr < shim[S][x].wc. This contradiction ensures that the CC has seen at least as many writes from S as have occurred locally at S, and obviously the CC cannot have

seen more writes from S than have occurred there, since S sends one update per write that it performs locally, so we know that CC.lCntr = shim[S][x].wc.
Therefore, we know that msg.ts = shim[S][x].ts$_{sync}$ + shim[S][x].wc + shim[S][x].wc_non_local + 1.
Therefore, when msg writes its timestamp to the shim cache, it will effectively increment the current timestamp. And, since shim[S][x].wc_non_local also gets incremented by msg, then we know that the final value of the shim timestamp with all updated values will be:
shim[S][x].ts′ = shim[S][x].ts$_{sync}$ + shim[S][x].wc + shim[S][x].wc_non_local′, so our claim holds in this case.
- Case 2: msg does not write its data. Then it increments the timestamp, and also increments shim[S][x].wc_non_local. By the IH, the old shim[S][x].ts is equal to shim[S][x].ts$_{sync}$ + shim[S][x].wc + shim[S][x].wc_non_local. After incrementing both shim[S][x].wc_non_local and shim[S][x].ts, we preserve the following equality: shim[S][x].ts = shim[S][x].ts$_{sync}$ + shim[S][x].wc + shim[S][x].wc_non_local.

Therefore, in any case, shim[S][x].ts is equal to shim[S][x].ts$_{sync}$ + shim[S][x].wc + shim[S][x].wc_non_local.

Lemma (update order): all updates to the shim caches happen in $\preceq_{CC}$ order.

Pf. Consider shim S and address x, we show by induction that updates to address x in S's shim cache are ordered by $\preceq_{CC}$.
BC. Consider U_0 the first update to S's cache. Case on U_0.
- Case 1: RRESP. Case on U_1, the second update to S's cache.
  - Case 1: RRESP. Since messages from the shims accept in order at the CC, the RREQ associated with U_0 accepted at the CC before the RREQ associated with U_1. Since the CC decides $\prec_{CC}$, and never responds to RREQs with stale data, we know that the data given to the second RRESP is at least as new as that given to the first RRESP. Thus, $U_0 \preceq_{CC} U_1$.
  - Case 2: local write. The WRITE update sent by U_1 to the CC will arrive after the RRESP of U_0 updated the shim cache by our assumption that U_0 was the first cache update and U_1 was the second. Therefore, U_1 arrives to the CC after U_0 and updates the CC with newer data, meaning $U_0 \preceq_{CC} U_1$.
  - Case 3: write update. Then the write update carrying U_1 will have hit the CC after the RREQ associated with U_0. Otherwise, S would not yet have been a sharer of x and therefore would not have been sent the write update carrying U_1 in the first place. Therefore, $U_0 \preceq_{CC} U_1$.
- Case 2: local write. Case on U_1, the second update to S's cache.
  - Case 1: RRESP. Then x was evicted between updates. Each RRESP brings the most recent value at the CC at address x into the shim cache. The WRITE associated with the local write of U_0 will hit the CC before the RREQ associated with U_1 since messages accept at the CC in the order they were sent. Therefore, $U_0 \preceq_{CC} U_1$.
  - Case 2: local write. WRITEs accept at the CC in the order they were sent, meaning the CC will order the writes U_0 and U_1 such that $U_0 \prec_{CC} U_1$.
  - Case 3: write update. Since U_0 was the first write to S's cache, S is not registered as a sharer or x until U_0's WRITE message is processed by the CC. Therefore, the write update of U_1 must have hit the CC after U_0, otherwise it would not have been sent as a write update. Therefore, $U_0 \preceq_{CC} U_1$.

IH. Suppose $U_0 \preceq_{CC} U_1 \preceq_{CC} \ldots \preceq_{CC} U_n$ for $0 < n$.
IS. Want to show $U_n \preceq_{CC} U_{n+1}$. Case on U_{n+1}.
- Case 1: RRESP. Then x was evicted from S after U_n. The message associated with U_n (RREQ, local WRITE, or remote WRITE) hit the CC before x was evicted from, and then brought back into, S's cache, meaning it hit before the RREQ associated with U_{n+1}. Therefore, $U_n \preceq_{CC} U_{n+1}$.
- Case 2: local write. If U_n was brought into the shim cache by a RRESP, this case is the same as Case 1.2 of the base case. Otherwise, suppose U_n was brought in by a local or remote write update. Then we know that since U_{n+1} is written to S's cache before it even arrives to the CC, it must be newer than the write that is already in its shim cache. This is because if the cached write is a local write, all writes arrive at the CC in order, so the local write is before U_{n+1} in $\prec_{CC}$. Otherwise, if the cached write is a non-local write, then it must have already gone through the CC in order to have been sent to S as a write update. Therefore, U_{n+1} will be after it in $\prec_{CC}$ order.
- Case 3: write update. Case on how U_n was brought into the shim cache.
  - Case 1: RRESP. Then S became a sharer of x on the read associated with U_n. The rest of this case is the same as Case 1.3 of the base case.
  - Case 2: remote write. Then there must have been some RRESP or local write prior to U_n which initially brought a into the shim cache, and synced timestamps, shim[S][x].ts$_{sync}$. Call this update U_i for $i \leq n$. By definition, shim[S][x].wc local writes occurred between U_i and U_n. In order for U_n to have accepted and written its data to the

shim, it had to have hit hte CC after all shim[S][x].wc local writes by the induction hypothesis; otherwise it would have violated $\prec_{CC}$.
The acceptance condition is as follows: msg.ts = shim[S][x].ts + 1 - (shim[S][x].wc - msg.wCntr). If a message does not satisfy this condition then it cannot write its value to the shim cache.
With this acceptance condition being satisfied by U_{n+1}, we know that:
U_{n+1}.ts = shim[S][x].ts + 1 - (shim[S][x].wc - U_{n+1}.wCntr). By the timestamp invariant, we can plug in a new value for shim[S][x].ts:
U_{n+1}.ts = (shim[S][x].ts_{sync} + shim[S][x].wc + shim[S][x].wc_non_local) + 1 - (shim[S][x].wc - msg.wCntr).
We also know that U_{n+1}.ts is equal to shim[S][x].ts_{sync} + msg.wCntr + CC.rCntr + 1, where CC.rCntr is the number of remote writes (writes not from S) to x that the CC accepted between U_i and U_{n+1}. Plugging this in, we get:
shim[S][x].ts_{sync} + msg.wCntr + CC.rCntr + 1 = shim[S][x].ts_{sync} + shim[S][x].wc + shim[S][x].wc_non_local + 1 - shim[S][x].wc + msg.wCntr. Simplifying, we get:
CC.rCntr = shim[S][x].wc_non_local.
This means that all the remote writes that the CC saw had already arrived to the shim by the time U_{n+1} arrives, meaning $U_n \prec_{CC} U_{n+1}$.

– Case 3: local write. Suppose when S synced timestamps with the CC after becoming a sharer of x the most recent time, it became a sharer on some update U_i and it synced on timestamp shim[S][x].ts_{sync}. Since CC timestamps only get updated on write updates from the shims, we know that U_{n+1}.ts = shim[S][x].ts_{sync} + CC.lCntr + CC.rCntr + 1, for CC.rCntr = non-local writes (from shims other than S) that hit the CC between U_i and U_{n+1}, CC.lCntr the local writes from S that hit the CC between U_i and U_{n+1} (including U_i but not including U_{n+1}).
Assume for the sake of contradiction that U_{n+1} arrives to the CC before U_n (i.e. $U_{n+1} \prec_{CC} U_n$). This means the U_{n+1}.ts < U_n.ts, since timestamps are assigned based on the order in which writes arrive to the CC.
Since U_{n+1} successfully updated the shim cache, we know that the following acceptance condition must hold:
msg.ts = shim[S][x].ts + 1 - (shim[S][x].wc - msg.wCntr). We can plug in our value for msg.ts and use Lemma (timestamp invariant) to give us:
shim[S][x].ts_{sync} + CC.lCntr + CC.rCntr + 1 = shim[S][x].ts_{sync} + shim[S][x].wc + shim[S][x].wc_non_local + 1 - (shim[S][x].wc - msg.wCntr). Simplifying (using the fact that msg.wCntr and CC.lCntr are by definition the same value), we get:
CC.rCntr = shim[S][x].wc_non_local.
Therefore, U_{n+1}.ts = shim[S][x].ts_{sync} + shim[S][x].wc_non_local + CC.lCntr + 1, and U_n.ts = shim[S][x].ts_{sync} + CC.lCntr + shim[S][x].wc_non_local. However, this means that U_n.ts + 1 = U_{n+1}.ts, meaning U_{n+1}.ts > U_n.ts.
But this contradicts our assumption that U_{n+1}.ts < U_n.ts. So by contradiction we have shown that $U_n \prec_{CC} U_{n+1}$.

Therefore, by induction we have shown that all updates to the shim caches happen in $\preceq_{CC}$ order.

Lemma (Per-Location Sequential Consistency): for all addresses x, there exists $\prec_x$ a total order on all writes to x, such that for all shims S, and any pair of reads $R0 \rightarrow_{\text{sb}} R1$ of S which read values $W0$ and $W1$, $W0 \preceq_x W1$.

Pf. We prove this claim by proving that $\prec_{CC}$, the order in which write updates hit the CC, is our desired order $\prec_x$.

Consider a shim S and address x. Assume for the sake of contradiction that the order of some reads violates $\prec_{CC}$, meaning there are two reads $R0$ and $R1$ of x at the shim, where $R0 \rightarrow_{\text{sb}} R1$, $R0$ reads some write $W0$ and $R1$ reads $W1$, but $W1 \prec_{CC} W0$. In the case of MemGlue$_{\text{U}}$, the values $W0$ and $W1$ returned to reads $R0$ and $R1$ may not necessarily be the values most recently placed into S's cache. Case on how $R0$ and $R1$ read the values they did.

- Case 1: Both read from cache updates (RRESPs or WRITE updates). Then by Lemma (update order), we are done.
- Case 2: Both read from the shim buffer. According to buffer reading rules, they will both read the value in the buffer such that: (1) the entry is newer than the newest local write (i.e. msg.wCntr = shim[X][a].wc), (2) the entry has the highest timestamp in the buffer and shim cache, and (3) there are no same-address reads awaiting a RRESP.
  By Condition (2), if $W0$ is still in the buffer when $R1$ performs, then we know that $R1$ will read $W0$ or newer, so $W0 \preceq_{CC} W1$. Otherwise, suppose $W0$ was popped from the buffer before $R1$ could perform. Assume for the sake of contradiction that $W1 \prec_{CC} W0$, meaning $W1$ was sent before $W0$ from the CC. By MemGlue's buffer-popping rules, messages get popped from the buffer in the order they were sent from the CC, meaning $W1$ would had to have been popped from the buffer before $W0$, which was popped from the buffer before $R1$ performed. However, this contradicts our assumption that $R1$ reads $W1$ from the shim buffer. So in either case, $W0 \preceq_{CC} W1$.
- Case 3: $R0$ reads from the buffer, but $R1$ does not. Case on how $R1$ reads its value.
  – Case 1: $R1$ reads from the shim cache.

When $R0$ reads from the buffer, it increments shim[S][x].rfBuf, signifying that some buffer message of address a has been read from. Case on the strength of $R1$.

* Case 1: $R1$ is not a read relax. Then $R1$ cannot proceed until all messages to address a in the buffer that have been read from have been popped from the buffer (i.e. shim[S][x].rfBuf returns to 0). So $W0$ will have been popped by the time $R1$ performs. If $W0$ was written to the shim cache, we know it was written before $W1$ (otherwise $R1$ would not have read from $W1$. Then by Lemma (update order), we can conclude that $W0 \preceq_{CC} W1$. Otherwise, if $W0$ did not update the shim cache, consider why. It must have been that at the time $W0$ was popped, $W0$.ts $\leq$ shim[S][s].ts. By the timestamp invariant, this means that $W0$.ts $\leq$ shim[S][x].ts_{sync} + shim[S][x].wc + shim[S][x].wc_non_local.
  We can break down $W0$.ts to know: shim[S][x].ts_{sync} + CC.lCntr + CC.rCntr + 1 $\leq$ shim[S][x].ts_{sync} + shim[S][x].wc + shim[S][x].wc_non_local, for CC.rCntr = non-local writes (from shims other than S) that hit the CC between W_{sync} and $W0$, CC.lCntr the local writes from S that hit the CC between W_{sync} and $W0$ (including W_{sync} but not including $W0$). Simplifying this, we get:
  CC.lCntr + CC.rCntr < shim[S][x].wc + shim[S][x].wc_non_local. Then it must either be the case that CC.lCntr < shim[S][x].wc, or CC.rCntr < shim[S].wc_non_local (if neither of those two conditions is true, then it cannot be true that CC.lCntr + CC.rCntr < shim[S][x].wc + shim[S][x].wc_non_local).
  If CC.lCntr < shim[S][x].wc, we know that a local write was made to S that was newer than $W0$, so $W0 \prec_{CC}$ shim[S][x].data.
  If CC.rCntr < shim[S].wc_non_local, then S saw a newer non-local write update than $W0$, meaning again that $W0 \prec_{CC}$ shim[S][x].data.
  In either case, at the time $W0$ was popped from the shim buffer, which is before $R1$ can read $W1$ from the shim cache, a newer value than $W0$ was already in the shim cache. Then by Lemma (update order), it must be the case that shim[S][x].data $\preceq_{CC} W1$, so it follows that $W0 \preceq_{CC} W1$.
* Case 2: $R1$ is a read relax. Then it will attempt to read from the message buffer, but by our assumption it will fail and will instead read from the shim cache. Since it was not able to read a write from the buffer early, this means that either no messages to address x were in the buffer anymore, or all the messages to address x failed one or more of the three conditions above. If no messages to address x are in the buffer anymore, this means that $W0$ was popped, and we can apply the same reasoning from the previous case to know that $W0 \preceq_{CC} W1$. Otherwise, suppose one of the conditions is violated for every message to address x in the buffer, including $W0$.
  - If Condition (1) is violated, then there is some more recent local write in the shim cache. Since msg.wCntr < shim[S][x].wc, we know that the write in the shim cache will reach the CC after $W0$ already hit. And since $R1$ reads this value from the shim cache, this value is $W1$. So $W0 \prec_{CC} W1$.
  - If Condition (2) is violated, then there is some message in the shim cache or in the shim buffer with a higher timestamp. Suppose Condition (1) is not violated, otherwise refer back to the previous case. Then we know that either the write in the shim cache has a higher timestamp than $W0$, or some write in the message buffer has a higher timestamp than $W0$. If the latter is true, then $R1$ would have read from that maximal-timestamp message instead of the shim cache, which violates our assumption. So suppose the shim cache entry has the highest timestamp.
    We know that the most recent entry in the shim cache is either a local write that hit the CC before $W0$ (so as not to violate Condition (1)), or it's a non-local write. We will start calling this value $W1$, as it is the write that will be read from by $R1$.
    Consider the first case, that $W1$ is a local write that hit the CC before $W0$. If it hit the CC before $W0$, this means it was already in the shim cache by the time $W0$ was sent to S, put in the shim buffer, and read from by $R0$. However, the same buffer reading conditions apply to both $R0$ and $R1$, meaning that if $R0$ chose to read $W0$ instead of $W1$, then $R1$ would also have chosen to read $W0$ since the values of shim[S][x].ts and shim[S][x].wc would have been the same. So this case is not possible.
    Now consider the second case, that $W1$ arrived as a write update and updated the shim cache. Consider the acceptance condition for $W1$ to write its value to the shim cache:
    msg.ts = shim[S][x].ts + 1 - shim[S][x].wc + msg.wCntr. Making the same substitutions as in previous cases (by breaking down msg.ts and by applying Lemma (timestamp invariant), we get: shim[S][x].ts_{sync} + CC.rCntr + CC.wCntr + 1 = shim[S][x].ts_{sync} + shim[S][x].wc + shim[S][x].wc_non_local + 1 - shim[S][x].wc + msg.wCntr. Simplifying in the usual way, we get:

CC.rCntr = shim[S][x].wc_non_local. We know by assumption that $W0$ is contributing to CC.rCntr but not shim[S][x].wc_non_local, because it was sent from the CC before $W1$ but has not been popped from the shim buffer yet, so it has not been counted in shim[S][x].wc_non_local. Therefore, the only way that CC.rCntr = shim[S][x].wc_non_local is if some write W_k updated the shim cache, and $W1 \prec_{CC} W_k$. However, this violated Lemma (update order). Therefore, $W1$ would not be able to write to the shim cache, contradicting our assumption that $W1$ updated the shim cache while $W0$ was still in the shim buffer.

· If Condition (3) is violated, then by MemGlue's specification, $R1$ will have to read from the CC, which is a different case than this, as in this case we assume that $R1$ reads from the shim cache.

– Case 2: $R1$ reads from the CC. Since $R0 \rightarrow_{sb} R1$ and $W0$ is in the shim buffer when $R0$ performs, and $R1$ reaches the CC after $R0$ performs, then $W0$ must have been in the CC by the time $R1$'s RREQ arrives. Therefore, $R1$ will read a value at least as new as $W0$. So $W0 \preceq_{CC} W1$.

- Case 4: $R1$ reads from the shim buffer and $R0$ reads from an update to the shim cache (either due to a RRESP, a local write, or a WRITE update). If $R0$ reads a local write then by Condition (1), $R1$ will read something newer. Otherwise, if $R0$ read a non-local write in the shim cache, then by Condition (2) $R1$ reads a value with a higher timestamp, meaning it hit the CC after the write in the shim cache. And since by Condition (1), msg.wCntr for the buffered message equals shim[S][x].wc, the higher timestamp implies that the buffered message is newer than the message in the shim cache. So $W0 \preceq_{CC} W1$.

Lemma (sb-cc): For $I \rightarrow_{sb} I'$ such that I and I' are writes, fences, or reads that retrieve their values via the CC, $I \prec_{CC} I'$.

Pf. Per MemGlue's design, messages are tagged with standard message counters, such that the CC may only accept messages in the order that they are sent from by the shims. And shims process instructions (and therefore send messages to the CC) in sb order. Therefore, I will hit the CC before I'.

Lemma (rf-cc): For any R, W such that $W \rightarrow_{rf} R$, W must have hit the CC before R either hit the CC, or completed locally.

Pf. Case on how some R read its value.

- Case 1: From the CC. If a read reads a value from the CC, then a RRESP message is sent back to the shim with this data. Clearly, the data carried by the RRESP will have to have hit the CC before the RREQ message arrived, because the data in the CC is what services the RREQ. The data carried by the RRESP will be used to service the original read by MemGlue's protocol design, so the claim holds.
- Case 2: From the shim cache/buffer. In order for the write to have arrived at the shim, it has to have gone through the CC and then been sent to the shim as a write update. Therefore, it hit the CC before the read completed locally.

Lemma (hb-cc): $I \rightarrow_{hb} I' \implies I \prec_{CC} I'$, for I and I' writes or fences.

Pf. By induction on the number of writes/fences in the hb chain after I.

BC: $I \rightarrow_{sb} I'$. By Lemma (sb-cc), $I \prec_{CC} I'$.

BC: $I \rightarrow_{sw} I'$. Then I' is a fence by the definition of sw. In order for I to be related to I' by sw, there must be a read R such that $R \rightarrow_{sb} I'$, and $W' \rightarrow_{rf} R$ for W' the write associated with I. By the definition of associated write, $I = W'$ or $I \rightarrow_{sb} W'$. In the latter case, by Lemma (sb-cc), $I \prec_{CC} W'$. By Lemma (rf-cc), W' must be in the CC before R completes, and by the design of MemGlue, I' cannot be sent to the CC until R completes. Therefore, $I \prec_{CC} I'$.

IH: Suppose for an hb chain containing n writes/fences $I_0, ..., I_n$, such that $I \rightarrow_{hb} I_0 \rightarrow_{hb} ... \rightarrow_{hb} I_n$, and $I \prec_{CC} I_n$.

IS: Consider the following structure: $I \rightarrow_{hb} I_0 \rightarrow_{hb} ... \rightarrow_{hb} I_n \rightarrow_{hb} I_{n+1}$. We want to show that $I \prec_{CC} I_{n+1}$.

If I_n is a write, then by the IH, $I \prec_{CC} I_n$. If I_n is a fence, then let W' be the write associated with I_n. By the IH $I \prec_{CC} I_n$, and by MemGlue's design, instructions are sent to the CC in program order, so $I_n \prec_{CC} W'$, so $I \prec_{CC} W'$.

We just established that $I \prec_{CC} W'$. Now, case on the specific hb edge type between I_n and I_{n+1}.

- Case 1: sb. Then by MemGlue's protocol design, writes and fences hit the CC in sb order, so $I_n \prec_{CC} I_{n+1}$. Therefore, $I \prec_{CC} I_{n+1}$.
- Case 2: sw. Then I_{n+1} must be a fence in order to be the sink of a sw edge. By Lemma (rf-cc), the read associated with I_{n+1} does not complete until after W' hits the CC, and I_{n+1} cannot be sent to the CC until after its associated read completes, by MemGlue's design. Therefore, $I \prec_{CC} W' \prec_{CC} I_{n+1}$, so $I \prec_{CC} I_{n+1}$.

Therefore, by induction, we have proven our lemma.

Lemma (rf-hb-cc): $W \rightarrow_{rf} R \rightarrow_{hb} I \implies W \prec_{CC} I$, for I a write or fence.

Pf. By induction on the number of writes/fences in the hb chain after R.
BC: $W \rightarrow_{\texttt{rf}} R \rightarrow_{\texttt{sb}} I$. By Lemma (rf-cc), by the time R completes, W will already be in the CC. And since sb-related instructions hit the CC in sb order, I doesn't hit the CC until after R completes. So $W \prec_{CC} I$.
BC: $W \rightarrow_{\texttt{rf}} R \rightarrow_{\texttt{sb}} I' \rightarrow_{\texttt{sw}} I$. By the reasoning from the previous case, $W \prec_{CC} I'$. And by Lemma (hb-cc), $I' \prec_{CC} I$. So $W \prec_{CC} I$.
IH: Suppose that if $W \rightarrow_{\texttt{rf}} R \rightarrow_{\texttt{hb}} I_0 \rightarrow_{\texttt{hb}} \ldots \rightarrow_{\texttt{hb}} I_n$, for $I_{0 \leq i \leq n}$ writes or fences and $n \geq 1$, then $W \prec_{CC} I_n$.
IS: We want to show that if $W \rightarrow_{\texttt{rf}} R \rightarrow_{\texttt{hb}} I_0 \rightarrow_{\texttt{hb}} \ldots \rightarrow_{\texttt{hb}} I_n \rightarrow_{\texttt{hb}} I_{n+1}$, then $W \prec_{CC} I_{n+1}$. By the induction hypothesis, we know that $W \prec_{CC} I_n$. And by Lemma (hb-cc), we know $I_n \prec_{CC} I_{n+1}$. So $W \prec_{CC} I_{n+1}$.

Lemma (sw-seen-id): if $(W_0 \rightarrow_{\texttt{hb}} W_1 \vee W_0 \rightarrow_{\texttt{rf}} R \rightarrow_{\texttt{hb}} W_1)$, for W_0 and W_1 writes, W_1 happening at some shim S, and the hb chain only containing reads and writes (no fences), then after W_1 performs at the CC, $\texttt{CC.seenIds[S][addr}(W_1)\texttt{].seenId} \geq W_0$.
Pf. By induction on the number of writes in the hb chain. We separate the different components of the precondition of the lemma (i.e. $(W_0 \rightarrow_{\texttt{hb}} W_1 \vee W_0 \rightarrow_{\texttt{rf}} R \rightarrow_{\texttt{hb}} W_1)$) into different base cases for clarity.
BC 1: $W_0 \rightarrow_{\texttt{sb}} W_1$. When W_0 performs at the CC, per MemGlue's definition it will update CC.seenIds[S].seenPerShim to be W_0. Then when W_1 performs at the CC, it will carry with it some seen id, call this msg.sID. It will update the value of $\texttt{CC.seenIds[S][addr}(W_1)\texttt{].seenId}$ to be $\texttt{max(msg.sID, CC.seenIds[S].seenPerShim)}$. By MemGlue's definition, the value of CC.seenIds[S].seenPerShim gets updated when a WRITE or RREQ from S is performed at the CC, and it gets updated to the newest write id assigned to the WRITE in the former case, or the max of the current CC.seenIds[S].seenPerShim, the write of the write being read from, and the last write id read at shim S. In any of these cases, it is clear that CC.seenIds[S].seenPerShim monotonically increases. Therefore, CC.seenIds[S].seenPerShim must be at least W_0 by the time W_1 performs at the CC, so $\texttt{CC.seenIds[S][addr}(W_1)\texttt{].seenId} \geq W_0$ after W_1 performs at the CC.
BC 2: $W_0 \rightarrow_{\texttt{rf}} R \rightarrow_{\texttt{sb}} W_1$. R may read from W_0 in one of three ways.

1. R retrieves the data from the CC. Then a RRESP is returned to S carrying W_0. Per MemGlue's specification, whether the RRESP accepts on time or early, it will add W_0 to shim[S].seenSetCache or shim[S].seenSetBuf.
2. R reads from the shim cache. Then a write update was sent to S and accepted. Per MemGlue's specification, whether the WRITE accepts on time or early, it will add W_0 to shim[S].seenSetCache or shim[S].seenSetBuf.
3. R reads from the shim buffer. Then W_0 will be added to shim[S].seenSetBuf.

In any case, W_0 will be in either shim[S].seenSetCache or shim[S].seenSetBuf. Then, when W_1 is later sent to the CC, it will send the maximum write ID in either of the two seen sets as its seen ID, call this msg.sID, with its update to the CC. Again, $\texttt{CC.seenIds[S][addr}(W_1)\texttt{].seenId}$ is updated to be the maximum of msg.sID and CC.seenIds[S].seenPerShim, so it must be the case that $\texttt{CC.seenIds[S][addr}(W_1)\texttt{].seenId} \geq W_0$ after W_1 hits the CC.
BC 3: $W_0 \rightarrow_{\texttt{sw}} R_0 \rightarrow_{\texttt{sb}} W_1$. Realize that we already proved this case in BC 1 under the second precondition.
IH: Suppose for any hb chain containing n writes for $n > 1$, our claim holds.
IS: Consider a chain of $n+1$ writes, such that $W_0 \rightarrow_{\texttt{hb}} W_n \vee W_0 \rightarrow_{\texttt{rf}} R \rightarrow_{\texttt{sb}} W_0' \rightarrow_{\texttt{hb}} W_n$. Case on the presence of an sw edge in the hb chain.

- Case 1: If there are no sw edges, then we know that $W_0 \rightarrow_{\texttt{sb}} W_{n+1}$ under the former precondition, and $W_0' \rightarrow_{\texttt{sb}} W_{n+1}$ under the latter, and we have already proven these cases in BC 1 and 2.
- Case 2: If there is at least one sw edge in the hb chain, then we have: $W_0 \rightarrow_{\texttt{hb}} W_i \rightarrow_{\texttt{sw}} R_i \rightarrow_{\texttt{sb}} W_n \vee W_0 \rightarrow_{\texttt{rf}} R \rightarrow_{\texttt{sb}} W_0' \rightarrow_{\texttt{hb}} W_i \rightarrow_{\texttt{sw}} R_i \rightarrow_{\texttt{sb}} W_n$. By the IH, we know that $\texttt{CC.seenIds[S']addr}(W_i)\texttt{].seenId} \geq W_0$, where S' is the shim that W_i takes place. Now case on how R_i reads from W_i.
  - Case 1: From the CC. Then the seen id that the RRESP associated with R_i will be the seen id of the last write to addr(R_i), which we know by the IH is at least W_0. When the RRESP returns to S, it will put the write id of the write that it read from (W_i) into either shim[S].seenSetCache or shim[S].seenSetBuffer, depending on it the RRESP accepts early. By Lemma (hb-cc), we know that the write id of W_0 is less than that of W_i. Therefore, when W_n sends its update to the CC, then it will carry the maximum seen id in the seen sets, which must be at least W_0. As before, this will guarantee that $\texttt{CC.seenIds[S][addr}(W_n)\texttt{].seenId}$ will be at least W_0.
  - Case 2: From the shim cache. Then W_i was sent to S as an update and updated the shim cache. Then, W_i will again end up in either shim[S].seenSetCache or shim[S].seenSetBuf, which by the same reasoning as the previous case will guarantee that $\texttt{CC.seenIds[S][addr}(W_n)\texttt{].seenId}$ will be at least W_0.

Therefore, by induction, we have proven our claim.

Lemma (hb-seen-sw): $(W \rightarrow_{\texttt{hb}} I_0 \rightarrow_{\texttt{sw}} I_1 \vee W \rightarrow_{\texttt{rf}} R_0 \rightarrow_{\texttt{hb}} I_0 \rightarrow_{\texttt{sw}} I_1) \wedge W \rightarrow_{\texttt{seen}} I_0 \implies W \rightarrow_{\texttt{seen}} I_1$.
Pf. Suppose I_0 performs at some shim S, and I_1 performs at some shim S'. Note that there are two pieces of the seen definition: 1) $\forall R$ such that R is a read, $I = R$ or $I \rightarrow_{\texttt{sb}} R$, and addr($R$) = addr($W$), $W \preceq_{CC}$ val(R), and 2) $\forall WF'$ such that WF' is a write or fence, and $I = WF'$ or $I \rightarrow_{\texttt{sb}} WF'$, or $I = WF'$, $W \prec_{CC} WF'$. Thus, for each case we will be proving two claims, which

we will refer to as seen_1 and seen_2.

$\underline{\mathsf{seen}_2}$: Consider some write or fence $WF_{S'}$ at S' such that $I_1 \rightarrow_{\mathsf{sb}} WF_{S'}$. So, we know that $W \rightarrow_{\mathsf{hb}} I_0 \rightarrow_{\mathsf{sw}} I_1 \vee W \rightarrow_{\mathsf{rf}} R_0 \rightarrow_{\mathsf{hb}} I_0 \rightarrow_{\mathsf{sw}} I_1$. We case on these two parts of the precondition.

- $W \rightarrow_{\mathsf{hb}} I_0 \rightarrow_{\mathsf{sw}} I_1$. Then I_1 is either equal to $WF_{S'}$, or $I_1 \rightarrow_{\mathsf{sb}} WF_{S'}$. Realize that there is an hb chain between W and $WF_{S'}$ in either case, so by Lemma (hb-cc), $W \prec_{CC} WF_{S'}$. This proves seen_2.
- $W \rightarrow_{\mathsf{rf}} R_0 \rightarrow_{\mathsf{hb}} I_0 \rightarrow_{\mathsf{sw}} I_1$. I_1 is either equal to $WF_{S'}$, or $I_1 \rightarrow_{\mathsf{sb}} WF_{S'}$. Realize that $I_0 \rightarrow_{\mathsf{hb}} WF_{S'}$ in either case, so by Lemma (hb-cc), $I_0 \prec_{CC} WF_{S'}$. We also know by our assumption that $W \rightarrow_{\mathsf{seen}} I_0$ and the definition of seen that $W \prec_{CC} I_0$. Combining this reasoning, we know that $W \prec_{CC} WF_{S'}$, proving seen_2.

$\underline{\mathsf{seen}_1}$: Now we prove seen_1. Consider some read R at S' such that $I_1 = R$ or $I_1 \rightarrow_{\mathsf{sb}} R$ and addr(R) = addr(W). First, case on whether S' is caching addr(W).

Case 1: S' is not caching addr(W) at the time R performs. Then R will have to retrieve its value from the CC. We break up the rest of this case based on which precondition we are assuming (i.e. $W \rightarrow_{\mathsf{hb}} I_0 \rightarrow_{\mathsf{sw}} I_1$ or $W \rightarrow_{\mathsf{rf}} R_0 \rightarrow_{\mathsf{hb}} I_0 \rightarrow_{\mathsf{sw}} I_1$).

- Case 1: $W \rightarrow_{\mathsf{hb}} I_0 \rightarrow_{\mathsf{sw}} I_1$. Notice that I_0 is either a write or fence because it is the source of an sw edge, so by Lemma (hb-cc), $W \prec_{CC} I_0$.
- Case 2: $W \rightarrow_{\mathsf{rf}} R_0 \rightarrow_{\mathsf{hb}} I_0 \rightarrow_{\mathsf{sw}} I_1$. Case on the hb edge.
  - Case 1: sb. Then by Lemma (rf-cc), W hit the CC before R_0 completed, and by MemGlue's specification, R_0 completed before I_0 could perform. So $W \prec_{CC} I_0$.
  - Case 2: hb edge contains some sw edge. Then $W \rightarrow_{\mathsf{rf}} R_0 \rightarrow_{\mathsf{sb}} I' \rightarrow_{\mathsf{sw}} I'' \rightarrow_{\mathsf{hb}} I_0 \rightarrow_{\mathsf{sw}} I_1$. By Lemma (hb-cc), $I' \prec_{CC} I_0$. And by Lemma (rf-cc), W hit the CC before R_0 completed, and by MemGlue's specification, R_0 completed before I' could hit the CC. So $W \prec_{CC} I_0$.

In any case, $W \prec_{CC} I_0$. And by Lemma (rf-cc), the write associated with I_0 (which is a either I_0 itself or a write after it in sb order) hit the CC before the read associated with I_1 completed. Since R either equal to said read or is after it in program order, then W was already in the CC by the time R went to retrieve its value via a RREQ. Since the CC decides the order of writes to an address, it only will have overwritten W with newer writes, so $W \preceq_{CC}$ val(R), and seen_1 is upheld.

Case 2: S' became a sharer of addr(W) only after W hit the CC, meaning no update of W is on the way. If addr(W) began being shared on a read, then by the reasoning of the prior case, the value pulled into the shim cache will be at least as new as W, and by Lemma (update order), no older value will overwrite it. So when R later happens at S', it will be the case that $W \preceq$ val(R). If addr(W) began being shared on some write $W_{S'}$, then because $W_{S'}$ is sb-after I_1, then $W \rightarrow_{\mathsf{hb}} W_{S'}$, and by Lemma (hb-cc), $W \prec_{CC} W_{S'}$. Again by Lemma (update order), R will not be able to read a value older than $W_{S'}$, so $W \prec_{CC}$ val(R). In either case, seen_1 holds.

Case 3: S' was already a sharer or addr(W) when W hit the CC, and is still a sharer of addr(W) at the time R occurs, meaning in this case we know that an update of W is on the way to S'. Now, case on whether or not the hb chain between W or R_0 and I_0 contains a fence. In the following cases, we prove that W must have arrived and been accepted at S' before I_1 can complete, and we later show how this implies seen_1.

- Case 1: It does contain a fence F. First, case on the precondition to show that F's FREQ hit the CC after W.
  - Case 1: $W \rightarrow_{\mathsf{hb}} F$. By Lemma (hb-cc), we know that F's FREQ hit the CC after W because $W \rightarrow_{\mathsf{hb}} F$.
  - Case 2: $W \rightarrow_{\mathsf{rf}} R_0 \rightarrow_{\mathsf{hb}} F$. Case on the hb edge.
    - Case 1: sb. Then by Lemma (rf-cc), W hit the CC before R_0 completed, and R_0 completed before F performed by MemGlue's specification, so $W \prec_{CC} F$.
    - Case 2: hb contains some sw edge. So $W \rightarrow_{\mathsf{rf}} R_0 \rightarrow_{\mathsf{sb}} I' \rightarrow_{\mathsf{sw}} I'' \rightarrow_{\mathsf{hb}} F$. By Lemma (hb-cc), $I' \prec_{CC} F$, and reusing the same reasoning as the previous case, $W \prec_{CC} I'$. So $W \prec_{CC} F$.

  Then, the update of W to S' will be sent before the FREQ associated with F, and it will have to accept at S' before F's FREQ, because FREQs cannot accept early at the shims by MemGlue's specification. Now case on how the read associated with I_1 (call this R_1, which is either equal to R or before it in sb order) reads its value.
  - Case 1: From the CC. Then its RREQ will arrive at the CC after the write associated with I_0, otherwise it wouldn't be able to read that value. Then the RRESP will be sent back to S', and will be ordered after the fence F that we assumed exists along the hb chain. By Lemma (fence order), this RRESP cannot be reordered with F's FREQ, so W must be processed at S' before R_1 can complete.
  - Case 2: From the shim cache. Then the write W_0 associated with I_0 must have updated the shim cache so that R_1 could have read from it. Since W_0 is equal to or sb-after I_0, then we know that $W \rightarrow_{\mathsf{hb}} W_1$, and therefore $W \prec_{CC} F \prec_{CC} W_1$. Therefore, the update of W is ordered before F's FREQ on the way to S', and the update of W_1 is ordered after it. By

Lemma (fence order), and by the fact that FREQ's arrive in order, this ensures that W is processed at S' before W_1. Therefore, once R_1 completes, we know that W has already been processed at S'.
  - Case 3: From the shim buffer. Then it must be the case that R_1 is a relaxed read, and I_1 is a fence. By Lemma (hb-cc), we know that $W \prec_{CC} F \prec_{CC} I_1$, so the FRESP associated with I_1 will be issued by the CC after W's update and F's FREQ. By Lemma (fence order), this FRESP cannot arrive before the FREQ, and since FREQ's cannot arrive early, the update of W must arrive before the FREQ. Therefore, by the time I_1 completes, W must already be processed at S'.
- Case 2: It does not contain a fence. Note that in this case, I_1 can still be a fence, but I_0 cannot. So I_0 is a $W_{rel/sc}$. By Lemma (sw-seen-id), we know that `CC.seenIds[S][addr(I₀)].seenID` will be at least W. Now case on I_1.
  - Case 1: I_1 is $R_{acq/sc}$. There are two ways that I_1 could read I_0:
    1. From the CC. Then the RRESP of I_1 carries `CC.seenIds[S][addr(I₀)].seenID` as its `msg.sID`, which is at least W, back to S'. And, because write ids only get put into `shim[s'].seenSetCache` once they get accepted in order (by MemGlue's design), we know that all messages that were dispatched prior to `msg.sID` will have already accepted. Therefore, since `msg.sID` $\geq W$, we know the write update of W will have to have accepted at S' before the RRESP of I_1 can be accepted and I_1 can complete.
    2. From shim cache. Then I_0 arrived as a write update and updated the shim cache. Again, the message would have carried some `msg.sID` such that `msg.sID` $\geq W$, since it would have been assigned `CC.seenIds[S][addr(I₀)].seenID` as its `sID` by the CC, per MemGlue's design. The write update cannot accept early unless `msg.sID` is in `shim[S'].seenSetCache`, and `msg.sID` cannot be in `shim[S'].seenSetCache` if the prior write update of W is still on the way per the same reasoning at Case 3.2.1.1. This means that W was accepted and is in the shim cache by the time I_1 completes.
  - Case 2: I_1 is F. Suppose R', the read associated with I_1, is relaxed, otherwise the same reasoning as Case 3.2.1 above applies. There are three ways that R' could have read from I_0:
    1. From CC. So when R' performs, a RREQ is sent to the CC, and a RRESP will be returned to S' that can accept early no matter what S' has seen (since the strength is relaxed). Suppose that when the RRESP accepts, addr(W) is in the shim cache but the update of W has not yet arrived (otherwise, per prior reasoning in Case 3.2.1, seen_1 clearly holds). Then, when I_1 performs, it sends a FREQ to the CC and an FRESP comes back to the shim. This FRESP must accept in order per MemGlue's design, so we know that the write update to W must have accepted by the time the FRESP is back and I_1 can complete.
    2. From shim cache. Then I_0 arrived as a write update and updated the shim cache. We use the exact same reasoning as Case 3.2.1.2 above.
    3. From the shim buffer. Then I_0 arrived as a write update and arrived early, being put into the shim buffer before it could be accepted. After R' reads from the shim buffer, it could be the case that the write update of W has not yet arrived. However, when I_1 is handled, it must go through the CC and come back via FREQ and FRESP messages. The FRESP cannot accept early, meaning the write update of W that was already in the network must accept first before I_1 can complete. Therefore, we know that when I_1 completes, W will be in the shim cache.

In any case, W must be processed at S''s cache before I_1 can complete, meaning 1) in the case that $I_1 = R$, then R must read a value at least as new as W by Lemma (update order), or 2) if $I_1 \rightarrow_{\mathsf{sb}} R$, then a value at least as new as W will be in the shim cache after I_1 is complete. By Lemma (update order), this means a value at least as new as W will be read by R, and $W \preceq_{CC}$ val(R) is upheld. Therefore, seen_1 holds for all cases above.

Therefore, both seen_1 and seen_2 hold, so $W \rightarrow_{\mathsf{seen}} I_1$.

Lemma (hb-seen): $W \rightarrow_{\mathsf{hb}} I \implies W \rightarrow_{\mathsf{seen}} I$.

Pf. By induction on the number of instructions in the hb chain. We again prove seen_1 and seen_2 separately as in Lemma (hb-seen-sw).

BC: $W \rightarrow_{\mathsf{sb}} I$. If seen_1 is not upheld, then there is some read R at shim S such that $W \rightarrow_{\mathsf{sb}} R$, but val($R$) $\prec_{CC} W$. However, R either read its value from S's cache, or from a RRESP. So either the RRESP updated S's cache after servicing R, or a prior write update updated S's cache, to later be read from by R. But by Lemma (update order), this is not possible for an update to contradict $\prec_{CC}$ order at S, so seen_1 is upheld.

If seen_2 is not upheld, then there is some write or fence WF' such that $I = WF'$ or $I \rightarrow_{\mathsf{sb}} WF'$, but $WF' \prec_{CC} W$. This also violates Lemma (update order), so seen_2 is upheld.

BC: $W \rightarrow_{\mathsf{sw}} I$. Consider the read R associated with I. We want to show $W \rightarrow_{\mathsf{rf}} R \implies W \rightarrow_{\mathsf{seen}} R$: this will prove that $W \rightarrow_{\mathsf{sw}} I \implies W \rightarrow_{\mathsf{seen}} I$. This is because I is either R, or some fence that is sb-after R, and both parts of our seen definition require properties to be true for all reads or writes that are sb-after the instruction I. Therefore, if we prove these properties for an instructions that is sb-before I, we prove the properties for I as well.

To prove seen_1, we can directly apply Lemma (PLSC): any read that is sb-after R must read a value at least as new as the value read by R, which is W. As for seen_2, by Lemma (rf-cc), W hit the CC before R completes. And by MemGlue's design, any write or fence that happens sb-after R will hit the CC after R completes. Therefore, $\forall WF'$ such that $R \rightarrow_{\mathsf{sb}} WF'$, $W \prec_{CC} WF'$, so seen_2 holds.
IH: Suppose for an hb chain with $n \geq 2$ instructions, $W \rightarrow_{\mathsf{hb}} I_n \implies W \rightarrow_{\mathsf{seen}} I_n$.
IS: Consider an $(n+1)$-length hb chain. We want to show that $W \rightarrow_{\mathsf{hb}} I_{n+1} \implies W \rightarrow_{\mathsf{seen}} I_{n+1}$. Case on the final edge.

- Case 1: sb. By the IH, $W \rightarrow_{\mathsf{seen}} I_n$. So for any read R s.t. $I_n = R$ or $I_n \rightarrow_{\mathsf{sb}} R$, seen_1 holds, and for any write or fence WF' s.t. $I_n = WF'$ or $I_n \rightarrow_{\mathsf{sb}} WF'$, seen_2 holds. These universal quantifications mean that these properties hold for I_{n+1} as well.
- Case 2: by Lemma (hb-seen-sw), this case is proven.

Therefore, by induction, we have proven our claim.

Lemma (rf-hb-seen): $R \rightarrow_{\mathsf{hb}} I \implies val(R) \rightarrow_{\mathsf{seen}} I$.

Pf. If R reads from initial memory, then we are trivially done. So suppose there is some write W such that $W \rightarrow_{\mathtt{rf}} R \rightarrow_{\mathsf{hb}} I$. We want to show that $W \rightarrow_{\mathsf{seen}} I$. We will again be proving seen_1 and seen_2. Let W be the shim on which R performs.

seen_2: By Lemma (rf-cc) we know that W is in the CC by the time R completes. Now case on the hb edge.

- Case 1: hb is only sb edges. Then $W \rightarrow_{\mathtt{rf}} R \rightarrow_{\mathsf{sb}} I$. By the definition of seen_2, we must show that for any write or fence WF' such that $I = WF'$ or $I \rightarrow_{\mathsf{sb}} WF'$, $W \prec_{CC} WF'$. Consider some such WF'. Then by Lemma (rf-cc), W is in the CC by the time R completes. And any instruction that is sb-after R cannot perform until R completes, meaning it won't be able to hit the CC until after W. Therefore, for this arbitrary WF' that is equal to or sb-after I, $W \prec_{CC} WF'$.
- Case 2: hb has some sw edge. Then $W \rightarrow_{\mathtt{rf}} R \rightarrow_{\mathsf{sb}} I' \rightarrow_{\mathsf{hb}} I$, where I' is the instruction on R's shim with an outgoing sw edge (so it must be a write or fence). By the reasoning in Case 1, $W \prec_{CC} I'$. And by Lemma (hb-cc), $I' \prec_{CC} I$. Therefore, $W \prec_{CC} I$. This means that for any write or fence WF' that is either equal to or sb-after I, since WF' cannot be sent to the CC until after I completes, $W \prec_{CC} WF'$.

Therefore, in either case, seen_2 holds.

seen_1: Consider some arbitrary read R' such that $I = R'$ or $I \rightarrow_{\mathsf{sb}} R'$. We want to show that $W \preceq_{CC} \mathrm{val}(R')$. We induct on the number of instructions in the hb chain.
BC 1: $W \rightarrow_{\mathtt{rf}} R \rightarrow_{\mathsf{sb}} I$. Then there must be some update to S's cache updating addr(W) to W, either because R retrieved the value from the CC, or it arrived as a WRITE update. Then by Lemma (update order), R' cannot read a value older than W. So seen_1 holds.
BC 2: $W \rightarrow_{\mathtt{rf}} R \rightarrow_{\mathsf{sb}} I' \rightarrow_{\mathsf{sw}} I$. By BC 1 and Lemma (hb-seen-sw), we know that $W \rightarrow_{\mathsf{seen}} I$.
IH: Suppose for an hb chain with $n \geq 2$ instructions, $W \rightarrow_{\mathtt{rf}} R \rightarrow_{\mathsf{hb}} I_n \implies W \rightarrow_{\mathsf{seen}} I_n$.
IS: Consider an $(n+1)$-length hb chain. We want to show that $W \rightarrow_{\mathtt{rf}} R \rightarrow_{\mathsf{hb}} I_{n+1} \implies W \rightarrow_{\mathsf{seen}} I_{n+1}$. Case on the final edge.

- Case 1: sb. By the IH, $W \rightarrow_{\mathsf{seen}} I_n$. So $\forall R$ s.t. R is a read, $I_n = R$ or $I_n \rightarrow_{\mathsf{sb}} R$, seen_1 holds. This universal quantification means that seen_1 holds for I_{n+1} as well.
- Case 2: by Lemma (hb-seen-sw), this case is proven.

Therefore, by induction, we have proven our claim.

1.2.3 Coherence Axiom.

Proof goal: if hb; eco is reflexive in some execution of a program, then MemGlue disallows the execution.
Pf. Consider some reflexive hb; eco edge involving $I1$ and $I2$: $I1 \rightarrow_{\mathsf{hb}} I2 \rightarrow_{\mathsf{eco}} I1$. Case on the eco edge:

- rf
  We have the following structure: $W \rightarrow_{\mathtt{rf}} R \rightarrow_{\mathsf{hb}} W$. By Lemma (rf-hb-cc), $W \prec_{CC} W$. This is clearly a contradiction because an instruction cannot hit the CC before itself.
- mo
  We have the following structure: $W1 \rightarrow_{\mathtt{mo}} W2 \rightarrow_{\mathsf{hb}} W1$. By Lemma (hb-cc), $W2 \prec_{CC} W1$. However, by mo order, which is decided by the CC, $W1 \prec_{CC} W2$, a contradiction.
- fr
  We have the following structure: $W \rightarrow_{\mathsf{hb}} R \rightarrow_{\mathtt{fr}} W$. By Lemma (hb-seen), $W \rightarrow_{\mathsf{seen}} R$. By the definition of seen, since addr(R) = addr(W), then $W \preceq_{CC} \mathrm{val}(R)$. However, by the definition of fr, $\mathrm{val}(R) \prec_{CC} W$, a contradiction.

- fr; rf
  We have the following structure: $R0 \rightarrow_{\texttt{fr}} W0 \rightarrow_{\texttt{rf}} R1 \rightarrow_{\texttt{hb}} R0$. By Lemma (rf-hb-seen), we know that since $R1 \rightarrow_{\texttt{hb}} R0$, and $R1$ reads from $W1$, then $W1 \rightarrow_{\texttt{seen}} R0$. By the definition of seen, this means that $W1 \preceq_{CC} \text{val}(R0)$. However, this contradicts the fr edge from $R0$ to $W1$, which would require that $\text{val}(R0) \prec_{CC} W1$.
- mo; rf
  We have the following structure: $R1 \rightarrow_{\texttt{hb}} W0 \rightarrow_{\texttt{mo}} W1 \rightarrow_{\texttt{rf}} R1$. By Lemma (rf-hb-cc), $W1 \prec_{CC} W0$. However, by mo order, $W0 \prec_{CC} W1$, a contradiction.

1.2.4 Atomicity Axiom.

We extend MemGlue in the same way as described in the ordered proof.

Proof Goal (Atomicity): if RMW intersected with (fr; mo) is non empty, or RMW intersected with eco is non empty for some execution, then the execution should not be observable in MemGlue.

Pf. We are trying to show that a LL-SC instruction pair cannot also be related by fr; mo. This means that if an LL instruction is related to its SC instruction by fr; mo, then the SC must not commit at the CC.

Suppose we have a LL/SC pair in MemGlue. Then in order for the LL to be fr-related to a write, the write must hit the CC after the LL, otherwise the LL would have read from that write. The LL will set the link register to x and the valid bit to 1, and the write of x will reset the valid bit to 0.

Then the SC hits the CC after the write, because they are related by mo. The link register will be cleared by then, so the SC will fail, and the RMW edge will not be present in the litmus test. In order for the LL/SC pair to perform and for the RMW edge to be added, the LL will need to be retried. But, when the LL is performed again, the LL and write of x will not be related by fr since the write is already in the CC by the time the second LL is performed. Then the litmus test would not have this fr; mo edge. Therefore, this behavior is not observable in MemGlue.

1.2.5 SC Axiom.

Edge Descriptions

- scb = hb|mo|fr
- psc_base = I1; scb; I2
  - I1 can be either some SC instruction, or $F \rightarrow_{\texttt{hb}} I$ for any instruction I.
  - I2 can be either some SC instruction, or $I \rightarrow_{\texttt{hb}} F$ for any instruction I.
- psc_fence = F; (hb|(hb; eco; hb)); F
- psc = psc_base|psc_fence

Note that we make a small deviation from the scb edge description in (cite RC11). RC11 weakened the scb relation because for Itanium processors, due to some intricacies with fences, it is not true that scb = (hb | mo | fr): if this were the case, there would be some programs that would be observable on Itanium processors, but would be fordibben by C11. Therefore, they weakened the scb edge, and thus the SC axiom. However, MemGlue treats scb as (hb | mo | fr), so we prove the SC axiom assuming scb = (hb | mo | fr). This means that MemGlue is stronger than RC11, which does not affect correctness.

Preliminary Claims

Lemma (hb-fr): if $I_{\text{n}} \rightarrow_{\texttt{hb}} R \rightarrow_{\texttt{fr}} W$, for I_{n} a fence or SC write, then W hits the CC after I_{n}.

Pf. Suppose R takes place at some shim S. Assume for sake of contradiction $W \prec_{CC} I_n$. Case on how R reads its value.

- Case 1: R reads from the CC. Case on the hb edge between I_n and R.
  - Case 1: hb = sb. Then since instructions hit the CC in sb order by the specification of MemGlue, the RREQ of R hits the CC after I_n, and therefore after W by our assumption. However, since the CC updates its cache lines in mo order, this means that R would have been returned a value at least as new as W in its RRESP. This contradicts the fr edge between R and W.
  - Case 2: hb contains after least one sw edge. Then there are some I', I'' such that $I_n \rightarrow_{\texttt{hb}} I' \rightarrow_{\texttt{sw}} R$, or $I_n \rightarrow_{\texttt{hb}} I' \rightarrow_{\texttt{sw}} I'' \rightarrow_{\texttt{sb}} R$. In the former case, we know by Lemma (hb-cc) that $I_n \prec_{CC} I'$, and by Lemma (rf-cc) that R's RREQ hit the CC after the write associated with I', meaning it hit the CC after I' by Lemma (sb-cc). Therefore, R hit the CC after W, meaning it will read a value W or newer, contradicting the fr edge. In the latter case, we know by Lemma

(hb-cc) that $I_n \prec_{CC} I'$, and by Lemma (rf-cc) that the read associated with I'' completed after the write associated with I' hit the CC. Therefore, combining with Lemma (sb-cc) we know that I' hit the CC before I'' completed locally, and since instructions are processed in sb order we know that R's RREQ didn't hit the CC until after I'' completed locally. Therefore, W must be in the CC before R's RREQ arrives, meaning R will read a value at least as new as W. This contradicts the fr edge between them.

- Case 2: Suppose R does not go through the CC, then it reads a value from the shim cache or message buffer. Note that we can assume that a write update of W was sent to R's shim – otherwise, it would be the case that addr(W) was not in R's shim cache by the time W happened. However, this would mean that another instruction on R's core brought addr(W) into the shim after W hit the CC, meaning the value brought into R's cache would be at least as new as W. This would contradict the fr edge between R and W. Therefore, we will assume that when W hit the CC, R's shim was a sharer and was sent a write update of W. Now case on the hb edge between I_n and R.
  - Case 1: hb = sb. Suppose I_n and R happen on some shim S. We know that a write update of W will be on the way to S before I_n reaches the CC, since it hit the CC before I_n by our assumption. Therefore, when I_n performs at S, since it is a W_{SC} or a fence, no other instruction at S may perform until it receives its acknowledgment back. The W update, having been sent before the acknowledgment, will already have arrived at S by the time I_n completes, and therefore that R that is sb-after I_n will eventually read a value at least as new as W (by Lemma (update order)). This would violate the fr edge between R and W.
  - Case 2: hb contains at least one sw, meaning we have some instructions I' and I'' such that $I_n \rightarrow_{\mathsf{hb}} I' \rightarrow_{\mathsf{sw}} R$ or $I_n \rightarrow_{\mathsf{hb}} I' \rightarrow_{\mathsf{sw}} I'' \rightarrow_{\mathsf{sb}} R$. Case on the existence of a fence between I_n and R.
    - Case 1: There is some fence $F1$ along this hb between I_n and R (meaning $F1$ may be I_n, I', or I'').
      If $F1$ is I'' or any instruction on S, then R cannot perform before I'' completes per MemGlue's specification. And, W hit the CC before the FREQ of I'', and since FRESPs arrive in order, W's update arrived at S before I'' received back its FRESP and therefore could complete. R cannot perform until I'' completes per MemGlue's specification, so W will already be in S's shim cache by the time R performs. By Lemma (update order), R must read a value at least as new as W, contradicting the fr edge between them.
      So suppose $F1$ performs at some shim other than S. If $F1$ is I_n, then by assumption $W \prec_{CC} F1$, otherwise by Lemma (hb-cc), $I_n \prec_{CC} F1$, meaning $W \prec_{CC} F1$. So in any case, $W \prec_{CC} F1$. When $F1$ reaches the CC, it will send out FREQ's to every shim, and FREQs may not arrive early. Since the W update arrived at the CC before the FREQ, it was sent to S before the return FREQ. Also, by Lemma (hb-cc), $F1 \preceq_{CC} I'$ (since $F1$ may be I', it is $\preceq_{CC}$ instead of $\prec_{CC}$). We also know that for the write associated with I', which we call W', $I' \preceq_{CC} W'$ by the definition of associated write, so $F1 \prec_{CC} W'$ (here, we use $\prec_{CC}$ instead of $\preceq_{CC}$ because $F1$ is a fence, so if $F1 = I'$, then it must be that $I' \prec_{CC} W'$, and if $F1$ is not I', then $F1 \prec_{CC} I' \preceq_{CC} W'$, meaning $F1 \prec_{CC} I'$). Case on how R'', the read associated with I'', reads its value.
      - Case 1: From the shim cache or message buffer, via a write update of W'. The write update of W' is sent from the CC after the FREQ of $F1$, since $F1 \prec_{CC} W'$.
      - Case 2: From the CC. Then the RRESP of R'' is ordered after W' at the CC because it read the value of W', meaning it must also be ordered after the FREQ of $F1$.

      In either case, by Lemma (fence order), this WRITE or W' or RRESP of R'' cannot arrive at S before $F1$'s FREQ, which must arrive after the update of W. Whether I' synchronizes with I'' or directly with R, W will be in S's shim cache before R can perform, meaning (by Lemma (update order)) that R will read a value at least as new as W. This contradicts the fr edge between R and W.
    - Case 2: the hb chain between I_n and R only contains writes and reads. Then we have some $W0$ and $R0$ such that: $I_n \rightarrow_{\mathsf{hb}} W0 \rightarrow_{\mathsf{sw}} R0$, and R is either $R0$ or is sb-after $R0$. Case on $R0$'s strength and how it reads its value.
      - Case 1: $R0$ is SC and reads from the CC. Then $R0 \neq R$, since we are in the case that R reads from the shim cache. We know by assumption that $W \prec_{CC} I_n$, and by Lemma (hb-cc) that $I_n \prec_{CC} W0$. Since $R0$ reads from $W0$, it will be ordered after it at the CC, so it will also be ordered after the W at the CC. Since SC instructions arrive in order, $R0$ will not receive its RRESP until after W arrives and updates the shim cache, meaning by Lemma (update order) that R will read a value at least as new as W. This contradicts the fr edge between them.
      - Case 2: $R0$ is a R_{acq} that reads from the CC. By assumption, $W \prec_{CC} I_n$, and by Lemma (hb-cc), $I_n \prec_{CC} W0$, meaning $W \prec_{CC} W0$. Then we know that the RREQ of $R0$ arrived after W hit the CC, since it must have hit the CC after $W0$ in order to read its value. Then, the only way that R can be related to W by fr is if W's update arrives out of order with $R0$'s RRESP, allowing R to read a value older than W.

We will show that this reordering is not possible by analyzing the seenId of the RRESP. First, realize that due to the case we are considering, the hb chain only contains reads and writes, and I_n may only be a fence or write SC, so it must be a write SC. Therefore, by Lemma (sw-seen-id), since $I_n \rightarrow_{\texttt{hb}} W0$, then CC.seenIds[S′][addr(W0)].seenId $\geq$ $\texttt{I}_\texttt{n}$, for S' the shim that $W0$ performs on. Also, $I_n \geq W$ by assumption. Therefore, when the RRESP of $R0$ is sent back to S, by MemGlue's specification we know that it will carry a seen id, called msg.sID, that is at least W. And it cannot accept early at S until some write id that is at least W is in the seenSetCache. By MemGlue's specification, the seenSetCache of S only gets updated with write ids that arrive in order. Therefore, since msg.sID is at least W, then W will have to have arrived in order and been put into the shim cache before the RRESP of $R0$ can be processed. Thus, W will already be in the shim cache by the time R is allowed to read a value from the shim cache, and by Lemma (update order), this value will be at least W. This contradicts the fr edge between R and W.

- Case 3: $R0$ reads from the shim cache. Then R can either be $R0$ or sb-after $R0$. We know that $W0$ hit R's shim cache before $R0$ performs, otherwise $R0$ would not have read from $W0$. Since $I_n \rightarrow_{\texttt{hb}} W0$, then by Lemma (hb-cc), $I_n \prec_{CC} W0$, meaning W would have also hit the CC before $W0$. Therefore, the only way to preserve the fr edge between R and W is to have $W0$'s update arrive out of order to S w.r.t. the write update of W. However, we will show that this is not possible by reasoning on the seen id of $W0$. Again realize that I_n must be a write, so by Lemma (sw-seen-id), CC.seenIds[S′][addr(W0)].seenId $\geq I_n$, which is at least W since W hit the CC before I_n. Then $W0$'s write update will carry a seen id, call this msg.sID, at least W. And the only way that $W0$ can accept early is msg.sID is already in S's seenSetCache, meaning a message with a write id at least msg.sID accepted in order at S. Therefore, we know that $W0$ cannot accept until W accepts and is written to the shim cache, since it will have to accept for msg.sID to be able to accept in order. Therefore, R (whether or not it is equal to $R0$) will read from a value at least as new as W. This contradicts the fr edge between them.

Therefore, in any case, we have proven our claim.

Lemma (psc-cc): $I_1 \rightarrow_{\texttt{psc}^*} I_n \implies I_i \prec_{CC} I_j$, for psc$*$ a chain of psc edges, $1 \leq i < j \leq n$ and $I_1 ... I_n$ writes or fences.

Pf. By induction on the number of writes and fences in the chain of psc edges.

BC 1: $I1$ and $I2$ are the only writes/fences, and psc$*$ is a single psc edge.

- Case 1: psc is psc_base. Case on the psc_base edge type.
  - mo. By the definition of mo, mo-related instructions hit the CC in mo order, so $I1 \prec_{CC} I2$.
  - hb. By Lemma (hb-cc), $I1 \prec_{CC} I2$.
  - fr. fr relates a read and write, and since $I1$ and $I2$ are writes or fences, a single fr edge between $I1$ and $I2$ is not possible.
- psc_fence. psc_fence can either be an hb edge or a hb; eco; hb edge. If psc_fence is an hb edge, then by Lemma (hb-cc), $I1 \prec_{CC} I2$. So consider psc = hb; eco; hb. Case on the eco edge.
  - rf. Then we have $I1 \rightarrow_{\texttt{hb}} W \rightarrow_{\texttt{rf}} R \rightarrow_{\texttt{hb}} I2$. By Lemma (hb-cc), $I1 \prec_{CC} W$, and by Lemma (rf-hb-cc), $W \prec_{CC} I2$, so $I1 \prec_{CC} I2$.
  - mo. Then we have $I1 \rightarrow_{\texttt{hb}} W0 \rightarrow_{\texttt{mo}} W1 \rightarrow_{\texttt{hb}} I2$. By the definition of mo, $W0 \prec_{CC} W1$, and by Lemma (hb-cc), $I1 \prec_{CC} W0$ and $W1 \prec_{CC} I2$. So $I1 \prec_{CC} I2$.
  - fr. Then we have $I1 \rightarrow_{\texttt{hb}} R \rightarrow_{\texttt{fr}} W \rightarrow_{\texttt{hb}} I2$. By Lemma (hb-fr), $I1 \prec_{CC} W$, and by Lemma (hb-cc), $W \prec_{CC} I2$. So $I1 \prec_{CC} I2$.
  - mo; rf. Then we have $I1 \rightarrow_{\texttt{hb}} W0 \rightarrow_{\texttt{mo}} W1 \rightarrow_{\texttt{rf}} R \rightarrow_{\texttt{hb}} I2$. By Lemma (hb-cc), the definition of mo, Lemma (rf-hb-cc), and the usual transitivity of $\prec_{CC}$, $I1 \prec_{CC} I2$.
  - fr; rf. Then we have $I1 \rightarrow_{\texttt{hb}} R0 \rightarrow_{\texttt{fr}} W \rightarrow_{\texttt{rf}} R1 \rightarrow_{\texttt{hb}} I2$. By Lemma (hb-fr), Lemma (rf-hb-cc), and transitivity of $\prec_{CC}$, $I1 \prec_{CC} I2$.

BC 2: $I1$ and $I2$ are the only writes or fence, but psc$*$ consists of a chain of intermediate reads. This means we have: $I1 \rightarrow_{\texttt{psc}} R_1 \rightarrow_{\texttt{psc}} ... \rightarrow_{\texttt{psc}} R_k \rightarrow_{\texttt{psc}} I2$. The only psc edge that relates two reads is hb, so this simplifies to: $I1 \rightarrow_{\texttt{psc}} R_1 \rightarrow_{\texttt{hb}} R_k \rightarrow_{\texttt{psc}} I2$. The first psc can only be hb (this is the only psc edge type relating writes or fences to reads), so we have: $I1 \rightarrow_{\texttt{hb}} R_k \rightarrow_{\texttt{psc}} I2$. Case on the remaining psc edge.

- Case 1: hb. Then our chain simplifies to $I1 \rightarrow_{\texttt{hb}} I2$, so by Lemma (hb-cc) we know that $I1 \prec_{CC} I2$.
- Case 2: fr. Then our chain is $I1 \rightarrow_{\texttt{hb}} R_k \rightarrow_{\texttt{fr}} I2$. By Lemma (hb-fr), we know that $I1 \prec_{CC} I2$.

In any case, we have proven our claim in the base cases.

IH: Suppose for a psc chain involving n write or fence instructions $I_1 \rightarrow_{\texttt{psc}} ... \rightarrow_{\texttt{psc}} I_n$, for any $1 \leq i < j \leq n$ $I_i \prec_{CC} I_j$.

IS: Consider a chain with $n+1$ write or fence instructions $I_1 \rightarrow_{\texttt{psc}} \ldots \rightarrow_{\texttt{psc}} I_n \rightarrow_{\texttt{psc}} I_{n+1}$. By the IH, we know all $I_1 \ldots I_n$ writes and fences hit the CC in psc order, so we need only show that $I_n \prec_{CC} I_{n+1}$. Case on how I_n and I_{n+1} are related.

- Case 1: I_n and I_{n+1} are related by a single psc edge. We have already proven this in BC 1.
- Case 2: psc$*$ is made up only of intermediate reads, i.e. $I_n \rightarrow_{\texttt{psc}} R_1 \rightarrow_{\texttt{psc}} \ldots \rightarrow_{\texttt{psc}} R_k \rightarrow_{\texttt{psc}} I_{n+1}$. We have also already proven this in BC 2.

Therefore, by induction, we have proven our claim.

Lemma (psc-reads): A psc cycle must have at least one write or fence.

Pf. A psc cycle consists of psc_base and psc_fence edges. psc_fence edges always occur between two fences, so a fence-free psc cycle will only have psc_base edges. psc_base edges consist of scb edges between two SC instructions. scb edges are either hb, mo, or fr edges. However, only hb edges can occur between two read instructions. So if we have a psc cycle containing only reads, then we know that $R_0 \rightarrow_{\texttt{hb}} \ldots \rightarrow_{\texttt{hb}} R_n \rightarrow_{\texttt{hb}} R_0$. Now case on the hb edges' types.

- Case 1: Every hb edge in this cycle is made up only of sb edges. Then we would have an sb cycle, which is not possible because MemGlue executes all instructions in sb order.
- Case 2: There is at least one sw edge in the hb cycle, i.e. between two reads in the psc cycle R_i and R_j, we have that $R_i \rightarrow_{\texttt{hb}} I1 \rightarrow_{\texttt{sw}} I2$, and either $I2 = R_j$ or $I2 \rightarrow_{\texttt{hb}} R_j$. Consider the write associated with $I1$, call this $W1$, and the read associated with $I2$, call this $R2$. Then $W1 \rightarrow_{\texttt{rf}} R2$. However, is either equal to $I2$, or $R2 \rightarrow_{\texttt{sb}} I2$, and $I2 \rightarrow_{\texttt{hb}} I1$, which is either equal to $W1$ or $I1 \rightarrow_{\texttt{sb}} W1$. Therefore, per the definition of hb we have that $R2 \rightarrow_{\texttt{hb}} W1$, and $W1 \rightarrow_{\texttt{rf}} R2$. Realize that this outcome violates the coherence axiom, and we have already proven that MemGlue does not allow program outcomes that violate coherence, so this outcome is not observable in MemGlue.

Therefore, a psc cycle that only contains reads is not observable in MemGlue, so a psc cycle must contain at least one write or fence.

Proof Goal (SC): if a program outcome exhibits a psc cycle, then the outcome is not observable in MemGlue.

Pf. Consider an arbitrary psc cycle. By Lemma (psc-reads), this cycle must contain at least one write or fence. Let $I1$ be the write or fence. Then by Lemma (psc-cc), all writes and fences in a psc chain hit the CC in psc order. However, since the chain is cyclic, this means that $I1$ would have to have hit the CC before itself, which is a contradiction.

1.2.6 No-Thin-Air Axiom.

Lemma (sb-rf-cc): Suppose $R0$ is related to $W1$ by (sb | rf), meaning a chain of sb and rf relations, and $R0$ reads from some write $W0$. Then $W0 \prec_{CC} W1$.

Pf. By induction.

BC: $W0 \rightarrow_{\texttt{rf}} R0 \rightarrow_{\texttt{sb}} W1$. By Lemma (rf-cc), we know that $W0$ hit the CC before $R0$ either hit the CC or completed locally. And $W1$ cannot hit the CC until after $R0$ performs because they are related by sb. So $W0 \prec_{CC} W1$.

IH: Suppose for a (sb|rf) chain starting with $R0$ that contains n writes, $W0 \prec_{CC} W_i$ for all $0 < i \leq n$.

IS: Consider a (sb|rf) chain starting with $R0$, which contains $n+1$ writes. By the IH, we know the n^{th} write hit the CC after $W0$. Case on the relation between W_n and W_{n+1}.

- Case 1: sb. Then W_n and W_{n+1} are from the same shim, and messages hit the CC in sb-order. Therefore, W_{n+1} hits the CC after W_n, and thus after $W0$.
- Case 2: rf; sb. Then W_n gets read from by some read R_n, which happens sb-before W_{n+1}. By Lemma (rf-cc), R_n either hits the CC or completes locally after W_n hits the CC. And W_{n+1} cannot hit the CC until after R_n has completed. Therefore, $W_n \prec_{CC} W_{n+1}$.

Therefore, $W0 \prec_{CC} W_{n+1}$.

Proof Goal (No-Thin-Air): if (sb | rf) is cyclic in some execution of a program, then the execution is not observable in MemGlue.

Pf (No-Thin-Air). Suppose (sb|rf) is cyclic. Then there must be at least one rf edge, because sb by definition cannot be cyclic. Then we know we have the following structure: $W0 \rightarrow_{\texttt{rf}} R0 \rightarrow_{(\texttt{sb}|\texttt{rf})+} W0$.

By Lemma (sb-rf-cc), we know that $W0$ will have to hit the CC before itself. This is not possible, meaning that if (sb|rf) is cyclic in a litmus test, then MemGlue will not let that test be observable.

1.3 MemGlue Tables

Msg Type	Cache state	Cache Action	Message to Network			Shim Action
WRITE A D	-	Perform write cache[A].TS++	ShimID	WRITE A D cache[A].TS	CC	-
READ A	V	Read cache[A].data				-
	I	-	ShimID	RREQ A _ cache[A].TS	CC	-
FENCE	-	-	ShimID	FREQ	CC	fencePending = true

Table 1. State transition table for Ordered MemGlue for requests from clusters to shims.

Msg Type	TS Guard	Cache Action	Shim Action
WRITE A D TS	TS > cache[A].TS	Perform write cache[A].TS = TS	-
	TS <= cache[A].TS	cache[A].TS++	-
WRITE_ACK A D TS	-	cache[A].TS = TS + cache[A].TS - 1 ? syncBit : cache[A].TS	syncBit = false pendingWSC = false
RRESP A D TS	-	Perform write cache[A].TS = TS	-
FRESP	-	-	fencePending = false

Table 2. State transition table for Ordered MemGlue for messages from the network to the shims.

Msg Type	Msg Strength	Sync Check	Cache Action	Message to Network		
				Src	Msg	Dst
WRITE S A D TS	RLX/REL	S already shares A	Perform write CC[A].TS++ CC[A].sharers += S	CC ... CC	WRITE A D CC[A].TS ... WRITE A D CC[A].TS	sharer 1 ... sharer n
		Otherwise	Perform write CC[A].TS++ CC[A].sharers += S	CC ... CC CC	WRITE A D CC[A].TS ... WRITE A D CC[A].TS WRITE_ACK A CC[A].TS	sharer 1 ... sharer n S
	SC	-				
RREQ S A _ TS	-	-	CC[A].sharers += S	CC	RRESP A CC[A].data CC[A].TS	S
FREQ	-	-	-	CC	FRESP	S

Table 3. State transition table for Ordered MemGlue for messages from shims to CC.

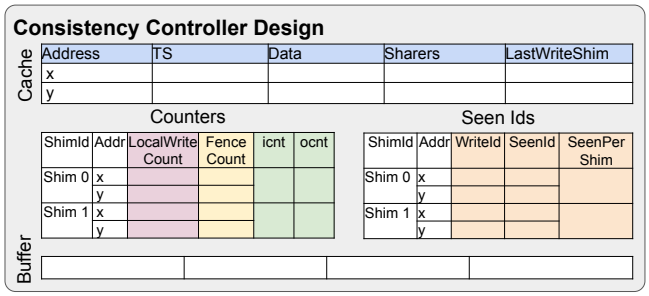
Msg Type	Stren	Addr in buffer	Cache State	Previous read pending	Cache Action	Message to Network			Shim Action
						Src	Msg	Dst	
WRITE A D	RLX REL	-	-	-	Perform write cache[A].TS++	ShimID	WRITE S A D cache[A].TS ocnt 0 max(seenSetCache,seenSetBuffer)	CC	ocnt++
	SC	-	-	-	Perform write cache[A].TS++	ShimID	WRITE S A D cache[A].TS ocnt 0 max(seenSetCache,seenSetBuffer)	CC	pendingWSC = trueocnt++
READ A	RLX	Yes			Return buffer.latest[A] ? cache[A].TS < buffer.latest[A] : cache[A]				seenSetBuf.append(buffer.lastest[A].wId) rfBuf = rfBuf + 1 ? ! buffer.latest[A].rf : rfBuf
		No	V	No	Return cache[A]				
	ACQ SC	-		Yes	-	ShimID	RREQ S A _ cache[A].TS ocnt 0 0	CC	ocnt++
			I	-					
FENCE	-	-	-	-	-	ShimID	FREQ ocnt	CC	fencePending = true ocnt++

Table 4. State transition table for Unordered MemGlue for cluster requests to shims.

Msg Type	Order Guard	Stren	Accept Early	TS Guard	Cache Action	Shim Action
WRITE S A D TS cnt fCnt wID sID wCntr	cnt == icnt	RLX REL SC	-	TS >cache[A].TS	Perform write cache[A].TS = TS	seenSetCache.add(wID) icnt++
				TS <= cache[A].TS	cache[A].TS++	icnt++
	cnt >icnt	RLX	Yes	TS >cache[A].TS	Perform write cache[A].TS = TS	seenSetBuffer.add(wID) buffer.append(1)
				TS <= cache[A].TS	cache[A].TS++	buffer.append(1)
			No	-	-	buffer.append(0)
		REL	Yes	TS >cache[A].TS	Perform write cache[A].TS = TS	seenSetBuffer.add(wID)
				TS <= cache[A].TS	cache[A].TS++	buffer.append(1)
			No	-	-	buffer.append(0)
		SC	-	-	-	buffer.append(0)
WRITE_ACK S A D TS cnt fCnt wID sID wCntr	cnt == icnt	-	-	-	cache[A].TS = TS = shim.wCntr - 1 ? syncBit : cache[A].TS	pendingWSC = false syncBit = false
	cnt >icnt	-	-	-	-	buffer.append(0)
RRESP S A D TS cnt fCnt wID sID wCntr	cnt == icnt	RLX ACQ SC	-	TS >cache[A].TS	Perform write cache[A].TS = TS	seenSetCache.add(wID) icnt++
				TS <= cache[A].TS	-	seenSetCache.add(wID) icnt++
	cnt >icnt	RLX ACQ	Yes	TS >cache[A].TS	Perform write cache[A].TS = TS	seenSetBuffer.add(wID) buffer.append(1)
				TS <= cache[A].TS	-	seenSetBuffer.add(wID) buffer.append(1)
			No	-	-	buffer.append(0)
		SC	-	-	-	buffer.append(0)
FREQ cnt	cnt == icnt	-	-	-	-	shim.fCnt++
	cnt >icnt	-	-	-	-	buffer.append(0)
FRESP	cnt == icnt	-	-	-	-	fencePending = false
	cnt >icnt	-	-	-	-	buffer.append(0)

Table 5. State transition table for Unordered MemGlue for messages from the network to the shims.

1.4 MemGlue$_U$ CC Design



1.5 Handling Counter Overflow

MemGlue must anticipate the eventual overflow of its many counters and preemptively reset the system. We propose that shims self-invalidate their caches once they send $(1/n * \text{counter\_max})$ messages to the CC. Then, the shim-local counters will reach at most $1/n * \text{counter\_max}$, and those at the CC will reach at most counter_max, meaning no overflow will occur. When the CC receives a self-invalidation of a shim cache, it should send out invalidation messages to the other shims. In response, each shim should invalidate all of its cached data and reset its counters. When the CC is notified that all local shims have invalidated, it can reset its counters, and the shims may resume processing their cluster-local accesses.

Msg Type	Order Guard	Cache Action	Stren	SeenId Action	Message to Network			CC Action
					Src	Msg	Dst	
WRITE S A D TS cnt sID	cnt == icnt[src]	Perform write cache[A].TS++ cache[A].sharers += src Set lastWrite	RLX	seenIds[src][A].wId = wIdCntr seenIds[src].seenPerShim = wIdCntr	CC ... CC	WRITE S A D cache[A].TS ocnt[sharer 1] seenIds[src][A].wId seenIds[src][A].sId ... WRITE S A D cache[A].TS ocnt[sharer n] seenIds[src][A].wId seenIds[src][A].sId	sharer 1 ... sharer n	icnt[src]++ wIdCntr++ ocnt[sharer i]++ for i = 1...n
			REL	seenIds[src][A].sId = max(sID,seenIds[src].seenPerShim) seenIds[src][A].wId = wIdCntr seenIds[src].seenPerShim = wIdCntr	 CC	If src not already sharing A: WRITE_ACK S A D cache[A].TS ocnt[src] seenIds[src][A].wId seenIds[src][A].sId	 src	If src not already sharing A: ocnt[src]++
			SC	seenIds[src][A].sId = max(sID,seenIds[src].seenPerShim) seenIds[src][A].wId = wIdCntr seenIds[src].seenPerShim = wIdCntr	CC ... CC CC	WRITE S A D cache[A].TS ocnt[sharer 1] seenIds[src][A].wId seenIds[src][A].sId ... WRITE S A D cache[A].TS ocnt[sharer n] seenIds[src][A].wId seenIds[src][A].sId WRITE_ACK S A D cache[A].TS ocnt[src] seenIds[src][A].wId seenIds[src][A].sId	sharer 1 ... sharer n src	icnt[src]++ wIdCntr++ ocnt[sharer i]++ for i = 1...n ocnt[src]++
	cnt >icnt[src]	-	-	-				buffer.append(0)
RREQ S A TS cnt sId	cnt == icnt[src]	cache[A].sharers += src Set lastWrite	RLX	-	CC	RRESP S A cache[A] cache[A].TS ocnt[src] lastWrite.wId lastWrite.sId	src	ocnt[src]++ icnt[src]++
			ACQ SC	seenIds[src].seenPerShim = max(seenIds[src][A].wId, seenIds[src].seenPerShim, lastWrite.sId)				
	cnt >icnt[src]	-	-	-				buffer.append(0)
FREQ	cnt == icnt[src]	-	-	-	CC	FREQ ocnt[shim i]	shim i	ocnt[shim i]++ icnt[src]++
	cnt >icnt[src]	-	-	-				buffer.append(0)

Table 6. State transition table for Unordered MemGlue for messages sent to the CC.