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. One instruction $I2$ has *seen* another instruction $I1$ ($I1 \rightarrow_{\text{seen}} I2$) if:

- If $I2$ goes through the CC, then $I1$ hit the CC before $I2$. If $I2$ performs at the shim without hitting the CC, then $I1$ is in the shim cache before $I2$ performs.
- If addr($I1$) is in the shim cache of the shim performing $I2$, then a value of addr($I1$) that is at least as new as $I1$ is in the cache by the time $I2$ completes.

Definition. eco = rf | mo | fr | mo ; rf | fr ; rf
Definition. $\prec_{CC}$ is the order in which write updates are processed at the CC.

Claim (update order): all updates to the shim caches happen in $\preceq_{CC}$ order.
Subclaim 1: each non-local write increments the local.ts by 1.
Pf (Subclaim 1). By induction.
BC: Suppose S synced on write i, and n local writes were performed before the first non-local write.

- Case 1: the first non-local write arrived at the CC after all n local writes. Then its timestamp is ts_{sync} + n + 1, and the local.ts is ts_{sync} + n. The the write can accept because its timestamp is greater than local.ts. This is correct because this write came after all local writes in $\prec_{CC}$.
- Case 2: the first non-local write arrived at the CC after m of the n local writes, and p of the local writes arrived at the CC after the first non-local write. So n = m + p, and p > 0. Then, we know that the timestamp of the first non-local write will be ts_{sync} + m + 1, and the local.ts will be ts_{sync} + m + p. Since p > 0, then the timestamp of the non-local write will be stale, and it will not write its data. However, it will increment the local.ts.

IH: Suppose n non-local writes have performed, and each has incremented the timestamp. Suppose m local writes performed before the n^{th} non-local write, and p after. Since local writes increment the timestamp, and the n previous non-local writes incremented the timestamp, then local.ts = ts_{sync} + n + m + p.
IS: Consider the $n+1^{th}$ non-local write W.

- Case 1: W arrived at the CC after all m + p non-local writes. Then its timestamp is set to ts_{sync} + n + m + p + 1. Then when it arrived to the shim, its timestamp is one greater than the local.ts, so it effectively increments the local.ts.
- Case 2: W arrived at the CC before some of the p non-local writes. Say p1 arrived before, and p2 arrived after, and p2 > 0. Then, W.ts = ts_{sync} + n + m + p1 + 1. And ts.local = ts_{sync} + n + m + p1 + p2, which is at least W.ts since p2 > 0. Therefore, W does not write its data, but it does increment the timestamp.

Therefore, in any case, non-local writes increment the local.ts.
Subclaim 2: the local timestamp of address x at shim S is equal to ts_{sync} + local.WC + (# non-local writes accepted). Local.WC represents the number of local writes performed (not including the write that initially synced with the CC).
Pf (Subclaim 2). Suppose that S synced on write i (meaning either the first read read from write i, or the first write to x performed at S had timestamp i). Then by Subclaim 1, non-local writes increment local.ts by 1, and we also know that all local writes increment the local.ts. Therefore, local.ts = ts_{sync} + local.WC + (# non-local writes accepted).
Pf (update order). Consider shim S and address a, we show by induction that updates to address a 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. Then a was evicted between updates. Each RRESP brings the most recent value and timestamp at the CC at address a into the shim cache. The RRESPs arrive to the CC in the order that the were sent by the network orderedness assumption, so it must be the case then that $U_0 \preceq_{CC} U_1$.
  - Case 2: local write. The WRITE update sent by U_1 to the CC will arrive after the RREQ associated with U_0 due to our network orderedness assumption. Therefore, U_1 arrives to the CC after U_0, 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 a 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 a was evicted between updates. Each RRESP brings the most recent value at the CC at address a 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 hit the CC in the order they were sent. Therefore, $U_o \preceq_{CC} U_1$.
  - Case 2: local write. WRITEs hit 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 a 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 a was evicted from S after U_n. The message associated with U_n (RREQ, local WRITE, or remote WRITE) hit the CC before U_n was 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} has not yet arrived at the CC at the time it is written to S, 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 W 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, W will be after it in $\prec_{CC}$.
- Case 3: write update. Case on how U_n was brought into the shim cache.
  - Case 1: RRESP. This case is the same as Case 1.3 of the base case.
  - Case 2: remote write. Write updates arrive in the order they were sent from the CC due to the orderedness of the network, so $U_n \prec_{CC} U_{n+1}$.
  - Case 3: local write. Suppose when S synced timestamps with the CC after becoming a sharer the most recent time, it synced on timestamp ts_{sync} (i.e. ts_{sync} is the timestamp at the CC when the WRITE or RREQ from S which made it a sharer arrived). We know that $U_{n+1}.\text{ts} > \text{ts}_{\text{sync}}$ since it only gets sent to S once S has already become a sharer, so let $U_{n+1}.\text{ts} = ts_{\text{sync}} + k + p + 1$ for k = non-local writes that happened between ts_{sync} and U_{n+1}, p the local writes that hit the CC between ts_{sync} and U_{n+1}.
    Since U_{n+1} wrote its data to the shim cache, it passed the timestamp check, so we know that $U_{n+1}.\text{ts} > U_n.\text{ts}$. And by Subclaim 2, we know that $U_n.\text{ts} = \text{ts}_{\text{sync}} + \text{local.WC} + (\#\text{ non-local writes})$. Plugging in the broken down values of U_n and U_{n+1}, we get $\text{ts}_{\text{sync}} + k + p + 1 > \text{ts}_{\text{sync}} + \text{local.WC} + (\#\text{ non-local writes})$. Since write updates arrive to the shims in the order they were sent, know that that $(\#\text{ non-local writes}) = k$. Simplifying the previous inequality given this knowledge, we know that $p + 1 > \text{local.WC}$. This means $p \geq \text{local.WC}$. Also, $p \leq \text{local.WC}$ since all local writes increment local.WC before they hit the CC, so $p = \text{local.WC}$. This means that no local writes happened at shim S after U_{n+1} hit the CC, i.e. the CC has seen all local writes at the time U_{n+1} is processed there, so U_{n+1} is newer than the value that is in the shim cache. This means $U_n \preceq_{CC} U_{n+1}$.

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

Claim (Per-Location Sequential Consistency): for all locations a there is a total order $\prec_a$ on all writes to a, and all reads at all shims honor $\prec_a$.
Pf. We prove this claim by proving that $\prec_{CC}$, the order in which write updates hit the CC, is our desired order $\prec_a$.

Consider a shim S and address a. AFSOC the order of some reads violates $\prec_{CC}$, meaning there are two reads $R0$ and $R1$ of a at the shim, where $R0 \rightarrow_{sb} R1$, $R0$ reads some write $W0$ and $R1$ reads $W1$, but $W1 \prec_{CC} W0$. The values $W0$ and $W1$ returned to reads $R0$ and $R1$ will be the values most recently placed into S's cache, either by write updates or by RRESPs. By Claim

(update order), all updates to S at address a happen in $\preceq_{CC}$ order, so $W0 \preceq_{CC} W1$. This contradiction proves our claim.

Claim (RF Edges): 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.

- 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, once the RRESP is in the network going back to the shim, we know that the data that gets returned to the cluster by the shim for that read is the value that is carried by the RRESP. Therefore, the value that will be read in the end will be the one read from originally in the CC. 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.

Claim (SW Edge 1): for instructions $I1$ and $I2$, if $I1 \rightarrow_{sw} I2$, then $I1 \rightarrow_{seen} I2$.
Pf. Case on $I1$. Note that Ordered MemGlue makes no distinction in the handling of different write strengths, so we do not consider strengths:

- Write W1. Then $I2$ is either a read, or a read followed in program order by a fence. We know by the RF edges claim that $I1$ will hit the CC before the read performs. And if the read is followed by a fence, then the fence will only be able to go to the CC once the read has performed because instructions are handled in program order. Therefore, in either case, $I1$ hits the CC before $I2$, so the first part of the seen definition holds.
  As for the second part, we know that the read that was part of $I2$ will read from $I1$, meaning addr($W1$) will be in the shim cache and will be a value at least as new as $W1$ by the time $I2$ completes (since writes to the shims do not violate mo, per Claim update order), so the second part of the definition of seen holds.
- Fence. Then we have a fence followed in program order by a write $W1$, which gets read from in program order by a read. This read is either $I2$, or it is followed in program order by a fence, which is $I2$. either way, we know that the first fence hit the CC before the write, and the write hit the CC before the read happened by the RF edges claim. And if a fence occurred after the read in program order, it hit the CC after the read performed, so it also hit the CC after the first fence. Therefore, $I1$ hit the CC before $I2$ performed, so the first part of the seen definition holds.
  As for the second part, again, we know that the read that was part of $I2$ will read from $I1$, meaning addr($W1$) will be in the reading core's cache by the time $I2$ performs, and a value at least as new as $W1$ will be present, because writes that arrive at the shims do not violate modification order (Claim: update order). Therefore, the second part of the seen definition holds.

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

Claim (SW Edges 2): for write $W0$ and instructions $I1$ and $I2$, if $W0 \rightarrow_{\text{hb}} I1 \rightarrow_{\text{sw}} I2$, then $W0 \rightarrow_{\text{seen}} I2$.
Pf. Suppose there is some write $W0$ that was seen by $I1$. Since $I1$ is the source of a SW edge, then we know that it is either a write or a fence. Then we know that:

1. $W0$ arrived at the CC before $I1$.
2. If addr($W0$) is in S's cache, then a value of addr($W0$) that is at least as new as $W0$ is in the cache by the time $I1$ completes.

The first part of the seen definition ensures that $W0$ arrives at the CC before $I1$. And by Claim (SW Edge 1), we know that $I1$ was seen by $I2$, meaning $I1$ arrived at the CC before $I2$ performs. Therefore, $W0$ arrives at the CC before $I2$ performs, so the first part of the seen definition is satisfied.

Note that in the next part of the proof, we do not case on $I1$ and $I2$ being either writes and reads, or fences. This is because if $I1$ and $I2$ are fences, there must be a following write / prior read in program order by the definition of sw. It is sufficient to only consider the writes and reads because these establish the order in which $W0$ arrives to the synchronizing shim. And if $W0$ arrives to the shim before the read, then it must also arrive to the shim before a po-after fence.

We now prove that the second part of the seen definition holds. If addr($W0$) is not shared by $I2$'s shim at the time $I2$ performs, then by the definition of seen, we are done. So suppose $I2$'s shim shares addr(W0).

- Case 1: $I2$'s shim was a sharer of addr($W0$) when $W0$ was performed at the shim and remained a sharer of addr($W0$) up to $I2$. Then, a write update of $W0$ would have been sent to $I2$'s shim, and would have arrived before $I2$ performed. To show this, let's case on how $I2$ performs.

  - Case 1: I2 reads $I1$ from the shim. Then, since $W0$ arrived at the shim before $I2$, then its write update would have been sent to $I2$'s shim before the RRESP of $I2$. Since the network is ordered, this means that it would have arrived before the RRESP, and therefore before $I2$ completed.
  - Case 2: $I2$ reads $I1$ from the shim cache. Then, $I1$ arrived as a write update to $I2$'s shim. Again, since the network is ordered, and since $W0$ arrived at the CC before $I1$, we know that $W0$ arrived at $I2$'s shim before $I1$, and therefore before $I2$ performed.
- Case 2: addr($W0$) was brought into $I2$'s cache before $I2$ performed, and after $W0$ hit the CC. There are two cases for how it was brought in.
  - Case 1: on a write. Then this write would have hit the CC after $W0$, meaning it's newer than $W0$. So the value in $I2$'s shim cache is at least as new as $W0$, since writes to the shim cache don't contradict mo.
  - Case 2: on a read. Then the read read $W0$ or newer and put this value into the shim cache. Therefore, since write updates to the shim cache don't contradict mo, the value in the shim cache will be at least as new as $W0$.

Therefore, $I2$ will have seen $W0$.

Claim (hb;rf extended): Prove that if $W0 \rightarrow_{\text{rf}} R0 \rightarrow_{\text{hb}} I1$, for $I1$ a write or fence, then $W0$ hit the CC before $I1$.
Pf. By induction.
BC. hb = sb. By the RF edges claim, 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 that $R0$ happens sb-before some instruction $I2$ that synchronizes with $I1$, in this case a fence (since writes cannot be the sink instruction of a sw edge). By the RF Edges claim, 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, so $I2$ hits the CC after $W0$. We also know that there is some read $R1$ sb-before $I1$ that reads from $I2$ or a sb-future write, per the definition of sw, and by the RF Edges Claim we know that $I2$ 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 two cases for how $I1$ relates to I_{n}.

- Case 1: sb. Then we know I_{n} was processed at the CC before $I1$, 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 the RF edges claim, 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 the RF Edges claim, 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 $W0$.

Claim (hb;mo extended): Prove that if $W0 \rightarrow_{\text{hb}} I1$, for $I1$ a write or fence, then $W0$ hit the CC before $I1$.
Pf. By induction.
BC: hb = sb. Instructions that go through the CC arrive at the CC in sb-order, so our claim holds.
BC: hb = sw ; sb. This means that $W0$ is read from by a synchronizing instruction $I2$, which is sb-before $I1$. Then, call the read that reads-from $W0$ $R2$ (note: $R2$ may not be the sink of the sw instruction, the sink may be a fence, but the following reasoning holds either way). By the RF Edges claim, $R2$ was processed after $W0$ hit the CC, either because it read $W0$ from the CC or because $W0$ hit the CC and was sent to its shim as a write update. And $I1$ hits the CC after $R2$, or after $R2$ is processed at the shim, so it must hit the CC after $W0$.
IH: Suppose for an hb chain starting with $W0$, 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. WTS that $I1$ hits the CC after $W0$. We know that the n^{th} write/fence, I^{n}, hit the CC after $W0$. There are two cases for how $I1$ relates to I^{n}.

- Case 1: sb. Then we know I^{n} was processed at the CC before $I1$, so by transitivity, we know that $W0$ hit the CC before $I1$.
- Case 2: sw. Case on I^{n}.
  - Case 1: I^{n} is a write. Then $I1$ is a fence, and there is a read sb-before $I1$ that reads from I^{n}. By the RF Edges claim, I^{n} hit the CC before the read performed, and $I1$ hit the CC after the read performed because instructions perform in program order. Therefore, $I1$ hit the CC after I^{n}, and by the IH and transitivity, $I1$ hit the CC after $W0$.

  - Case 2: I^n is a fence. This means that I^n is followed by a write in program order, and this write is read from by a read sb-before $I1$, a fence. Again by the RF Edges claim, this write hits the CC before the read performs, meaning that the fence also hits the CC before the read performs. Since $I1$ is sb-after the read, it hits the CC after the read performs or is dispatched to the CC, meaning $I1$ hits the CC after I^n. By the IH and transitivity, then $I1$ hits the CC after $W0$.
- Case 3: sw ; sb. Case on I^n.
  - Case 1: I^n is a write. This means that I^n was read from by a read sb-before $I1$. By the RF edges claim, 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 $W0$.
  - Case 2: I^n is a fence. This means that I^n is followed by a write in program order, and this write is read from by a read sb-before $I1$. Again by the RF Edges claim, this write hits the CC before the read performs, meaning that the fence also hits the CC before the read performs. Since $I1$ is sb-after the read, it hits the CC after the read performs or is dispatched to the CC, meaning $I1$ hits the CC after I^n. By the IH and transitivity, then $I1$ hits the CC after $W0$.

Claim (sb reads): if $R0 \rightarrow_{sb} R1$, then $\forall$ instructions $I1$ s.t. $I1 \rightarrow_{seen} R0$, $I1 \rightarrow_{seen} R1$.
Pf. Suppose there is some write $W0$ s.t. $W0 \rightarrow_{seen} R0$. Then $W0$ hit the CC before any message associated with $R0$, and if addr($W0$) is in the shim cache of $R0$'s shim, then a value at least as new as $W0$ was in the cache.

- Case 1: addr(W0) was not in the shim cache at either R0 or R1. Then we are done by our definition of seen.
- Case 2: addr(W0) was brought into the cache between R0 and R1 (either because it wasn't in the cache at R0, or it was but was then evicted).
  - Case 1: $W0$ is brought into the cache via a write. Then, since $W0$ was seen by R0, which happened sb-before the write that brought addr(W0) back into the cache, this last write must have hit the CC after W0. Therefore, it has written newer data to the cache, which cannot be overwritten with stale data. So R1 will see a value at least as new as W0.
  - Case 2: $W0$ is brought back into the cache via a read. Then, R1 cannot perform until all prior instructions perform, so it must wait until the read that reads from addr(W0) completes. And since this prior read hits the CC after W0, it will read a value at least as new as $W0$ and put this value into the shim cache. Therefore, R1 will see a value of addr(W0) that is at least as new as W0, meaning it has seen W0.
- Case 3: addr(W0) was evicted after R0 and was not brought back into the cache. Then we are done by our definition of seen.
- Case 4: addr(W0) was never evicted. Then because writes to the shims do not contradict mo, a value at least as new as $W0$ must be in the shim cache. Therefore, R1 has seen W0.

Claim (hb Write Read): If $W0 \rightarrow_{hb} R0$, then $W0 \rightarrow_{seen} R0$.
Pf. By induction.
BC: hb = sb. Then $W0$ was written into the shim cache, and arrived at the CC before $R0$ (if $R0$ was sent to the CC). If addr($W0$) was evicted before $R0$ performed, then by the definition of seen, we are done. Otherwise, we know that a value at least as new as $W0$ is in the shim cache by Claim (Per-Location Sequential Consistency). So again by our definition of seen, we are done.
BC: hb = sw. Then $R0$ reads from $W0$. By the RF Edges claim, $W0$ must be in the CC before $R0$ arrives. And, when $R0$ reads $W0$, it will put this value in the shim cache (unless a newer value is already there). So by our definition of seen, $R0$ will have seen $W0$.
IH: Suppose for an hb chain involving n reads (where reads that initiate an sw relation going into a following fence are counted), each read has seen $W0$.
IS: Consider an hb-chain of n+1 reads. By the IH, the nth read has seen $W0$, meaning $W0$ was in the CC before any messages associated with the read arrived, and if addr($W0$) was in the shim cache of the nth read, then a value at least as new as $W0$ was in the cache. There are two cases for the relationship between R_n and $R0$:

- sb. This means $R_n \rightarrow_{sb} R0$. By Claim (sb reads), $R0$ has seen everything R_n saw, meaning $R0$ has seen W0.
- sb;sw. This means that for some write or fence WF, $R_n \rightarrow_{sb} WF \rightarrow_{sw} R0$. By the SW Edges 2 claim, we know that any write that happens hb-before an instruction I1 will be seen by an instruction I2 that synchronizes with it. Therefore, since $W0$ happens hb-before WF, it will be seen by $R0$.

Claim (hb;fr;rf): The following structure is not possible: $R1 \rightarrow_{hb} R0 \rightarrow_{fr} W1 \rightarrow_{rf} R1$.
Pf. By the definition of fr, there is some write $W0$ that is mo-before $W1$, which is read from by $R0$. Case on the hb edge.

- Case 1: hb = sb. Since $R1$ reads from $W1$, then $W1$ will have been in the shim cache by the time $R0$ happens. By Claim (Per-Location Sequential Consistency), $R0$ must read $W1$ or newer in order not to violate coherence, violating the assumed fr edge between them.
- Case 2: hb = hb ; sw ; sb. Then we have the following structure: $W1 \rightarrow_{\text{rf}} R1 \rightarrow_{\text{hb}} I1 \rightarrow_{\text{sw}} I2 \rightarrow_{\text{sb}} R0$. By Claim (hb;rf extended), $W1$ hit the CC before $I1$. This means that $W0$ and $W1$ are in the CC before $I1$ completes. If $I1$ is a fence, consider the write that is sb-after it that is read from by $I2$ (if $I2$ is a read; otherwise consider the read that is sb-before $I2$). Case on how the read reads from the write.
  - Case 1: From the shim cache. Then the write arrived as an update. Since $W1$ hit the CC before $I1$ and therefore before the write, if it is on its way as an update, it will arrive before the read completes. Then, $R0$ will read $W1$ or newer from the shim cache instead of $W0$. Otherwise, if addr($W1$) is not in the shim cache, then $R0$ will read its value from the CC; $W1$ will already be in the CC, so again it will read $W1$ or newer.
  - Case 2: From the CC. If addr($W1$) is in the shim cache, then it will already have been sent as an update before the synchronizing read sends its RREQ to the CC, so $W1$ will be in the shim cache before the read completes. Then $R0$ will read $W1$ or newer from the shim cache, a contradiction. Otherwise, $R0$ will send its RREQ to the CC after the read that is sb-before it, meaning it will observe $W1$ already in the CC and will read $W1$ or newer.

1.1.2 Coherence Axiom.

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

- rf
  Then I2 is a write that gets read from by I1. By Claim (hb;rf extended), we know that the write that I1 read from hit the CC before I2. However, I2 is the write that I1 read from, which means that I2 would had to have hit the CC before itself, a contradiction.
- mo
  By Claim (hb;mo extended) above, we know that all writes in an hb chain hit the CC after the writes that happen before them in the chain. Since the eco edge is mo, the two instructions connected by the hb edge in this case are writes. Therefore, the last write in the hb chain hits the CC after the first write. However, these writes are related in the opposite direction by mo, meaning the second write hit the CC before the first write, which is a contradiction.
- fr
  If a write W1 is related to a read R1 via fr, then the read read from a previous write $W0$ in mo. By Claim (hb Write Read) above, we know that R1 has seen W1. However, by the definition of seen, this means that the R1 would have read W1 or newer, not W0, because W1 or newer would either be in the shim cache of the reading core, or it would be in the CC for a RRESP to the CC to read. But then we would have a rf relation between these two instructions, not a fr, a contradiction.
- fr ; rf
  We have the following structure: $R1 \rightarrow_{\text{hb}} R0 \rightarrow_{\text{fr}} W1$, and R1 reads from W1. This structure is not observable by Claim (hb;fr;rf).
- mo ; rf
  We have the following structure: $R1 \rightarrow_{\text{hb}} W0 \rightarrow_{\text{mo}} W1$, and R1 reads from W1. By Claim (hb;rf extended) above, $W0$ will hit the CC after the W1. However, this contradicts mo order, which implies that $W0$ hit the CC before 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.
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 W 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 W 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 $F \rightarrow_{\text{hb}} I$ for any instruction I.
  - I2 can be either some SC instruction, or $I \rightarrow_{\text{hb}} F$ 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

Claim (hb;fr): if $I_{\text{n}} \rightarrow_{\text{hb}} R \rightarrow_{\text{fr}} W$, for I_{n} a write or fence, then W hits the CC after I_{n}.
Pf. AFSOC W arrives to the CC before I_{n}. Case on how R reads its value.

- Suppose R reads from the CC. By Claim (hb write read), we know that R has seen I_{n}. This means that I_{n} was already in the CC before R went to the CC, and since W arrived before I_{n}, then W is already in the CC cache by the time R arrives. Therefore, R will read a value at least as new as W, contradicting the definition of the fr edge between R and W.
- 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.
  - hb is a sequence of sb edges. We know that a write update of W will be on the way to the shim before I_{n} reaches the CC. Therefore, when I_{n} performs, since it is a W_{SC} or a fence, no other instruction may perform until it receives its acknowledgement back. The W update, having been sent before the acknowledgement, will already have arrived at the reading shim by the time I_{n} completes, and therefore that R will eventually read a value at least as new as W. This would violate the fr edge between R and W.
  - hb = sw + sb chain (at least one sw edge). Then we know that R must be sb-after some sink instruction $I1$ of a sw edge. Then we have the format: $I_{\text{n}} \rightarrow_{\text{hb}} I0 \rightarrow_{\text{sw}} I1 \rightarrow_{\text{sb}} R$. First, realize that if there is some fence $F1$ along this chain, then by Claim (hb;mo extended), In hit the CC before $F1$, meaning W hit the CC before $F1$. When $F1$ reaches the CC, it will send out FREQ's to every core, and messages to the cores may not get reordered across FREQs. Since the W update was sent before the FREQ, and the write that is read from by R's core is ordered after it, then W will arrive to R's shim before the synchronizing read on R's core is allowed to complete. Therefore, R would read W or newer, contradicting the fr edge.
    Therefore, consider an sb-sw chain with only reads and writes, no fences. Then we have: $I_{\text{n}} \rightarrow_{\text{hb}} W0 \rightarrow_{\text{sw}} R0 \rightarrow_{\text{sb}} R$. Case on the sw edge.

    - $R0$ misses at the shim cache. Then it will go through the CC and will be ordered after the W. Since messages arrive in order in Ordered MemGlue, $R0$ will not receive its RRESP until after W arrives, meaning R will read a value at least as new as W. This contradicts the fr edge.
    - $R0$ hits at the shim cache. We then know that $W0$ hit R's shim before $R0$ performs. Since $W0$ will have seen In by Claim (hb;mo extended), then I_n will have hit the CC before it, meaning W would have also hit the CC before it. Therefore, the W update will have been sent out before the $W0$ update, meaning by the time R0 reads from $W0$, W will be in R's shim cache, and R will read a value at least as new as W. This would contradict the fr edge between R and W.

Therefore, W must arrive to the CC after I_n.

Claim (psc W/F order): writes and fences in psc chains hit the CC in psc order (i.e. if $I1 \rightarrow_{psc^*} I2$, then $I1$ hits the CC before $I2$, for $I1$ and $I2$ writes or fences.
BC: 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.
  - hb. By Claim (hb;mo extended), $I1$ hits the CC before $I2$.
  - 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. If psc_fence is an hb edge, we are done by base case 1.2, so consider F;hb;eco;hb;F. Case on the eco edge.
  - rf. Then we have $F0 \rightarrow_{hb} W \rightarrow_{rf} R \rightarrow_{hb} F1$. By Claim (hb;mo extended), Claim (hb;rf extended), and transitivity, $F0$ hit the CC before $F1$.
  - mo. Then we have $F0 \rightarrow_{hb} W0 \rightarrow_{mo} W1 \rightarrow_{hb} F1$. By Claim (hb;mo extended), the definition of mo, and transitivity, $F0$ hit the CC before $F1$.
  - fr. Then we have $F0 \rightarrow_{hb} R \rightarrow_{fr} W \rightarrow_{hb} F1$. By Claim (hb;fr), Claim (hb;mo extended), and transitivity, $F0$ hits the CC before $F1$.
  - mo;rf. Then we have $F0 \rightarrow_{hb} W0 \rightarrow_{mo} W1 \rightarrow_{rf} R \rightarrow_{hb} F1$. By Claim (hb;mo extended), the definition of mo, Claim (hb;rf extended), and transitivity, $F0$ hit the CC before $F1$.
  - fr;rf. Then we have $F0 \rightarrow_{hb} R0 \rightarrow_{fr} W \rightarrow_{rf} R1 \rightarrow_{hb} F1$. By Claim (hb;fr), Claim (hb;rf extended), and transitivity, $F0$ hit the CC before $F1$.

BC: $I1 \rightarrow_{psc^*} I2$ for a chain of psc reads in between $I1$ and $I2$. WTS if $I1 \rightarrow_{psc^*} I2$ for the instructions in psc* all reads, then $I2$ hit after $I1$. Specifically, we have: $I1 \rightarrow_{psc} R1 \rightarrow \ldots \rightarrow_{psc} Rk \rightarrow_{psc} I2$. The only psc edge that relates two reads is hb, so this simplifies to: $I1 \rightarrow_{psc} R1 \rightarrow_{hb} Rk \rightarrow_{fr} I2$. The first psc can only be hb (that's the only edge relating writes or fences to reads), and the second psc edge can be either hb or fr. If both psc are hb, then we are done by Claim (hb;mo extended). So, consider the case: $I1 \rightarrow_{hb} Rk \rightarrow_{fr} I2$. By Claim (hb;fr), $I2$ hits the CC after $I1$.
IH: Suppose for a psc chain involving n W/F instructions, all W/F instructions hit the CC in psc order.
IS: Consider a chain with n+1 W/F instructions. By the IH, we know all $I1 \ldots I_n$ writes and fences hit the CC in order, so we need only show that In hits the CC before I_{n+1}. Case on how I_n and I_{n+1} are related.

- Case 1: WTS if $I_n \rightarrow_{psc} I_{n+1}$ for no intermediate reads, then I_{n+1} hit after I_n. We have already proven this in the first base case, so we are done.
- Case 2: WTS if $I_n \rightarrow_{psc^*} I_{n+1}$ for psc* made up only of intermediate reads, then I_{n+1} hit after I_n. We have also already proven this in the second base case, so we are done.

Claim (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. Only hb edges can occur between two read instructions. Case on the hb edges' types.

- All hb edges just consist of sb edges. Then we would have an sb cycle, which is not possible because MemGlue executes all instructions in sb order.
- There is at least one sw edge in the hb cycle. Consider the source and sink instructions of some sw edge, $I1$ and $I2$ ($I1 \rightarrow_{sw} I2$). If $I1$ is a write and $I2$ is a read, then $I1 \rightarrow_{rf} I2$, and $I2 \rightarrow_{hb} I1$. This violates the coherence axiom, which we have already proven the MemGlue does not do, so this is not observable in MemGlue. Now suppose $I1$ and $I2$ are

fences. Then we have $F1 \rightarrow_{\text{sb}} W \rightarrow_{\text{rf}} R \rightarrow_{\text{sb}} F2$. By the definition of hb, we still have the $R \rightarrow_{\text{hb}} W$, and $W \rightarrow_{\text{rf}} R$, so the coherence axiom is still violated. The cases where one of $I1$ or $I2$ is a fence follow the same reasoning.

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 psc has a cycle, then the behavior is not observable in MemGlue.
Pf. Consider an arbitrary psc cycle. By Claim (psc reads), this cycle must contain at least one write or fence. Let I1 be the write or fence. Then by Claim (psc W/F order), 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.

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

Claim (W;rf;R;sb|rf): 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 $W0$ hit the CC before $W1$.
Pf. By induction.
BC: $W0 \rightarrow_{\text{rf}} R0 \rightarrow_{\text{sb}} W1$. 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 $W1$ hit the CC after $W0$.
IH: Suppose for a (sb | rf) chain starting with $R0$, and contains n writes, each write hits the CC after $W0$.
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 $W_{\text{n+1}}$.

- sb. Then W_{n} and $W_{\text{n+1}}$ are from the same shim, and messages hit the CC in sb-order. Therefore, $W_{\text{n+1}}$ hits the CC after W_{n}, and thus after $W0$.
- rf; sb. Then W_{n} gets read from by some read R_{n}, which happens sb-before $W_{\text{n+1}}$. By the RF Edges claim, R_{n} either hits the CC or completes locally after W_{n} hits the CC. And $W_{\text{n+1}}$ cannot hit the CC until after R_{n} has completed. Therefore, W_{n} hits the CC before $W_{\text{n+1}}$.

Therefore, $W_{\text{n+1}}$ hits the CC after $W0$.

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

Definition. One instruction $I1$ has *seen* another instruction $I2$ if:

- If $I2$ goes through the CC, then $I1$ hit the CC before $I2$. If $I2$ performs at the shim without hitting the CC, then $I1$ is in the shim cache before $I2$ performs.
- If addr($I1$) is in S's cache, then a value of addr($I1$) that is at least as new as $I1$ is in the cache by the time $I2$ completes.

Definition. eco = rf | mo | fr | mo ; rf | fr ; rf

Claim (fence order): FREQs from the CC to the shims prevent messages that were sent after the FREQ from arriving before the FREQ.

Pf. Suppose a FREQ is sent from shim 1, and shim 2 shares data that shim 1 also shares. Suppose the fence arrives at the CC, then a FREQ will be sent to shim 2. Suppose some message $M2$ was sent to shim 2 after the fence. The CC counts fences, meaning that if the FREQ sent to shim 2 is the nth fence to go to that shim, then the fenceCount counter tagged onto $M2$ will be $\geq n$.

Now suppose $M2$ arrives at the shim before the FREQ. We know that the fenceCount at shim 2 is < n, since it hasn't seen the n^{th} FREQ yet, and it accepted FREQs in order. Then $M2$.fenceCount > shim.fenceCount, meaning that $M2$ will not be accepted at the shim (and cannot be accepted until at least the n^{th} fence has been accepted).

Claim (timestamp invariant) the local timestamp of address x at shim S is equal to ts_{sync} + local.WC + (# non-local writes accepted). Local.WC represents the number of local writes performed (not including the write that initially synced with the CC).

Pf. By induction. Suppose that X synced on write i, so ts_{sync} is i.

BC: No local or non-local writes have performed. Then local.ts = i + 0 + 0, which is correct. BC: n local writes have performed, and no non-local writes. Then each local write increments the timestamp by 1, local.ts = i + n, which is equal to ts_{sync} + local.WC + (# non-local writes accepted).

BC: one non-local write has accepted, no local writes. Then the timestamp of the non-local write is equal to i+1, which is equal to i + local.WC + (# non-local writes accepted). And it sets local.ts = i+1, so our claim holds.

BC: one non-local write has accepted, n local writes.

- Case 1: The non-local write reached the shim after all of the local writes. Then its timestamp is i+n+1. Then when it reaches the shim and is accepted, it sets local.ts to i+n+1 = i + local.WC + (#non-local writes accepted), so our claim holds.
- Case 2: The local write reached the shim before at least one of the local writes.Then its timestamp is $\leq$ i + n, so once it gets accepted at the shim, it will not be able to overwrite the data and timestamp. Instead, it will increment the local timestamp by 1, so local.ts = i + n + 1. Then our claim still holds.

IH: Suppose after each of n local and non-local writes, the timestamp at the shim is equal to i + local.WC + (# non-local writes accepted).

IS: Suppose an n+1^{th} write W_{curr} is performed at the shim.

- Case 1: The write is a local write. Then it increments the local.ts and local.WC, so it still holds that local.ts = i + local.WC + (# non-local writes accepted).
- Case 2: The write is a non-local write.
  - Case 1: The write arrived to the CC after all of the local writes that have been performed at the shim. Then we know that it has the highest timestamp that the CC has seen, i + n + 1. When it arrives at the shim, it sets the timestamp to i + n + 1, which is equal to i + local.WC + (# non-local writes accepted)
  - Case 2: The write arrived to the CC early. Suppose some write W_{prev} was the most recently accepted write, and by the IH we know that the timestamp after it was accepted was i + local.WC + (# non-local writes accepted).
    Suppose that the order that the writes hit the CC looks like this:
    W_{prev} [L local writes, m non-local writes] W_{curr} [p local writes].
    In other words, W_{prev} is followed by L local writes and m local writes, then W_{curr} arrives, then p local writes occur. Then by our acceptance logic, the following formula would have to be true for W_{curr} to accept:

msg.ts = W_{prev} + L + p + 1 - (p)
which is equivalent to W_{prev} + L + m + 1 = W_{prev} + L + 1,
which implies that m = 0.
So if W_{curr} gets accepted early, then there must have been 0 non-local writes between it and the previously accepted non-local write. But this is not true because we assumed that W_{curr} has arrived early, before other non-local writes. So W_{curr} cannot accept early, and therefore the timestamp will not yet be changed.
In order for W_{curr} to accept, it must either be accepted on time by outCounter, or msg.ts = local.ts + 1 - (local.WC - CC.WC). Suppose q additional local writes happen after W_{curr} gets buffered. Then we are waiting for the following to hold:
W_{prev} + L + m + 1 = local.ts + 1 - (q + p)
We know that local.ts = W_{prev} + L + m + (q + p) for W_{curr} to accept. Realize that this is equal to the number of non-local writes between W_{prev} and W_{curr}, plus the number of local writes since W_{prev}.

* If (q+p) is zero, then no local writes have performed since W_{curr} hit the CC. This means that local.ts = W_{prev} + L + m, which is equal to msg.ts - 1. So when W_{curr} accepts, it will set the timestamp to W_{prev} + L + m + 1. By our induction hypothesis, we know that W_{prev} = i + local.WC' + (# non-local writes accepted'). So adding all these constants together, we know that the current timestamp is equal to i + local.WV + (# non-local writes accepted).
* If (q+p) is non-zero, then some local writes have performed since W_{curr} hit the CC, and W_{curr}'s timestamp will be stale. Then by accepting W_{curr}, the timestamp will be incremented and will now be W_{prev} + L + m + (q+p) + 1, which again is equal to i + local.WC + (# non-local writes accepted).

Therefore, in any case, the local timestamp of address x at shim S is equal to ts_{sync} + local.WC + (# non-local writes accepted).

Claim (update order): all updates to the shim caches happen in $\preceq_{CC}$ order.
Pf (update order). Consider shim S and address *a*, we show by induction that updates to address *a* 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 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 RREQ associated with U_0 because messages are accepted at the CC in order. Therefore, U_1 arrives to the CC after U_0, 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 *a* 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 *a* was evicted between updates. Each RRESP brings the most recent value at the CC at address *a* 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 *a* 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 *a* was evicted from S after U_n. The message associated with U_n (RREQ, local WRITE, or remote WRITE) hit the CC before U_n was 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} has not yet arrived at the CC at the time it is written to S, 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 W 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, W will be after it in $\prec_{CC}$.

- Case 3: write update. Case on how U_n was brought into the shim cache.
  - Case 1: RRESP. 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, ts_{sync}. Call this update U_i for $i \leq n$. Suppose l 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 l local writes by the induction hypothesis; otherwise it would have violated $\prec_{CC}$.
    The acceptance condition is as follows: msg.ts = local.ts + 1 - (local.WC - msg.WC). 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 = local.ts + 1 - (local.WC - U_{n+1}.WC). By the timestamp invariant, we can plug in a new value for local.ts:
    U_{n+1}.ts = (ts_{sync} + l + local.NL) + 1 - (local.WC - msg.WC), where local.NL is the number of non-local writes to a that the shim has accepted since U_i.
    We also know that U_{n+1}.ts is equal to ts_{sync} + msg.WC + CC.NL + 1, where CC.NL is the number of remote writes to a that the CC accepted before U_{n+1}. Plugging this in, we get:
    ts_{sync} + msg.WC + CC.NL + 1 = ts_{sync} + l + local.NL + 1 - l + msg.WC. Simplifying, we get:
    CC.NL = local.NL.
    This means that all the non-local 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 the most recent time, it synced on timestamp ts_{sync} (i.e. ts_{sync} is the timestamp at the CC when the WRITE or RREQ from S which made it a sharer arrived). We know that U_{n+1}.ts > ts_{sync} since it only gets sent to S once S has already become a sharer, so let U_{n+1}.ts = ts_{sync} + $(l-1)$+ CC.NL +1 for CC.NL = non-local writes that happened between ts_{sync} and U_{n+1}, l the local writes that hit the CC between ts_{sync} and U_{n+1}.
    AFSOC U_{n+1} arrives to the CC before U_n (i.e. $U_{n+1} \prec_{CC} U_n$. Plugging in the values to the acceptance condition (see prior case), we get:
    ts_{sync} + $(l-1)$ + CC.NL + 1 = ts_{sync} + l + 1 + local.NL - $(l-(l-1))$. Simplifying, we get:
    CC.NL = local.NL
    Therefore, U_{n+1}.ts = ts_{sync} + local.NL + $(l-1)$ + 1, and U_n.ts = ts_{sync} + l + local.NL. However, these simplify to the same value, so U_n.ts = U_{n+1}.ts. But this would mean that U_{n+1} would not have been able to update the shim cache with its value, because only write updates whose timestamps exceed the current shim timestamp may write. This contradicts our assumption the U_{n+1} updated the shim cache. 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.

Claim (Per-Location Sequential Consistency): for all locations a there is a total order $\prec_a$ on all writes to a, and all reads at all shims honor $\prec_a$.

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

Consider a shim S and address a. AFSOC the order of some reads violates $\prec_{CC}$, meaning there are two reads $R0$ and $R1$ of a at the shim, where $R0 \rightarrow_{sb} R1$, $R0$ 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 Claim (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 read (i.e. msg.WC = shim[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 $W1$ is still in the buffer when $R2$ performs, then we know that $R2$ will read $W1$ or newer, so $W1 \preceq_{CC} W2$. Suppose that $W1$ has been popped from the buffer. AFSOC $W2 \prec_{CC} W1$. But then $W2$ must have a lower ocnt than $W1$ since it was sent first from the CC. $W1$ can only be popped once all messages with a lower ocnt have accepted, contradicting our assumption that $W1$ was already popped. So either way, $W1 \preceq_{CC} W2$.
- Case 3: $R1$ reads from the buffer, buf $R2$ does not. Case on how $R2$ reads its value.
  - Case 1: $R2$ reads from the shim cache.

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

* Suppose $R2$ is not a read relax. Then $R2$ cannot proceed until all messages to address a in the buffer that have been read from has been popped from the buffer. So $W1$ will have been popped. If $W1$ was written to the shim cache, then by Claim (update order), $W1 \preceq_{CC} W2$. If not, consider why. It must have been that at the time $W1$ was popped, $W1.ts \leq$ shim[a].ts. By the timestamp invariant, this means that $W1.ts \leq$ ts_{sync} + local.WC + #non-local write updates received at the shim. We can break down $W1$.ts to know:
  ts_{sync} + CC.WC + #non-local writes received at the CC + 1 $\leq$ ts_{sync} + local.WC + #non-local writes received at the shim. Simplifying this, we get:
  CC.WC + #non-local writes received at the CC < local.WC + #non-local writes received at the shim. Then it must either be the case that CC.WC < local.WC, or $W1$ occurred after fewer non-local writes updates than the value in the shim cache. If the first case, we know that a local write was made to the shim that we newer than $W1$, so $W1 \prec_{CC}$ shim.value. In the second case, the shim saw a newer non-local write update than $W1$, meaning again that $W1 \prec_{CC}$ shim.value. By Claim (update order), it follows that $W1 \preceq_{CC} W2$.
* Otherwise, if $R2$ is a read relax, then it will attempt to read from the message buffer. If it succeeds, this case will be covered by Case 2 above. Otherwise, it will read from the shim cache. However, if it was not able to read a write from the buffer early, this means that either no messages to address a were in the buffer anymore, or all the messages to address a failed one or more of the three conditions above. If no messages to address a are in the buffer anymore, this means that $W1$ was popped, and we can apply the same reasoning from the prior case to know that $W1 \preceq_{CC} W2$.
  Otherwise, suppose one of the conditions is violated for every message to address a in the buffer, including $W1$.
  - If Condition (1) is violated by the message containing $W1$, then there was a local write that performed that was newer than $W1$. So $R2$ will read a newer value than $W1$, and $W1 \preceq_{CC} W2$.
  - If Condition (2) is violated, then there is some message in the shim cache or in the shim buffer with a higher timestamp. Suppose the shim cache entry has a higher timestamp, then by the same reasoning used for when $R2$ is not a read relax, the entry in the shim cache must be newer than $W1$, and $R2$ will read from this value, so $W1 \preceq_{CC} W2$. Otherwise, there is some message in the message buffer Wm with a maximal timestamp. But by our assumption, this message also violates one of the conditions. If it violates (2), then the entry in the shim cache has a higher timestamp than it. Again by the reasoning from when $R2$ is not a read relax, the shim cache entry must be newer than Wm and this is what $R2$ reads, so $W1 \prec_{CC} Wm \prec_{CC} W2$. If it violates (3) then $W1$ would also violate (3), so suppose it violates Condition (1). By then Condition (1) would also be violated for $W1$. Otherwise, this would imply that $W1 \prec_{CC} Wm$, but there exists some local write WL such that $Wm \prec_{CC} WL \prec_{CC} W1$. So in any case, Condition (1) or (3) will be violated, so this case can be completed by the proofs of those cases.
  - If Condition (3) is violated, then $R2$ will have to read from the CC, which is a different case than this.

– Case 2: $R2$ reads from the CC.
  Since $R1 \rightarrow_{sb} R2$ and $W1$ is in the shim buffer when $R1$ performs, and $R2$ reaches the CC after $R1$ performs, then $W1$ must have been in the CC by the time $R2$'s RREQ arrives. Therefore, $R2$ will read a value at least as new as $W1$. So $W1 \preceq_{CC} W2$.

- Case 4: $R2$ reads from the shim buffer and $R1$ reads from an update to the shim cache (either due to a RRESP, a local write, or a WRITE update). If $R1$ reads a local write then by Condition (1), $R2$ will read something newer. Otherwise, if $R1$ read a non-local write in the shim cache, then by Condition (2) $R2$ reads a value with a higher timestamp. And since by Condition (1), the write count of the buffered message equals the local write count, the higher timestamp implies that the buffered message is newer than the message in the shim cache. So $W1 \preceq_{CC} W2$.

Claim (RF Edges): 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.

- 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, once the RRESP is in the network going back to the shim, we know that the data that gets returned to the cluster by the shim for that read is the value that is carried by the RRESP (see MemGlue structures claim above). Therefore, the value that will be read in the end will be the one read from originally in the CC. 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.

Claim (SW Edge 1): for instructions $I1$ and $I2$, if $I1 \rightarrow_{sw} I2$, then $I1 \rightarrow_{seen} I2$.
Pf. Suppose there is some write $W0$ that was seen by I1. Then we know that:

1. $W0$ arrived at the CC before $I1$, or if they are from the same core, $W0$ was processed / sent to the CC before $I1$, and
2. If addr($W0$) is in S's cache, then a value of addr($W0$) that is at least as new as $W0$ is in the cache by the time $I1$ completes.

We also know that the seenSet of $I1$'s shim cache at the time it performs contains a write that is $\geq W0$, by Claim (hb edges) below (which states: along an hb chain, each synchronizing read reads a write at least as new as the first write in the chain).

The first part of the seen definition ensures that $W0$ arrives at the CC before $I1$. And if $I1$ gets read from, that means that $I2$ must have arrived at the CC before the read that reads from it arrives at the CC / is processed within the shim.

Case on the sw edge:

- Case 1: $I1$ is W_{rel} or W_{SC}. Now case on $I2$:
  - Case 1: $I2$ is R_{acq} or R_{SC}. There are two ways that $I2$ could read $I1$:
    1. From the CC. Then $I2$ must have arrived to the CC after $I1$ to have read it, and $W0$ arrived before both of those. The return message is: RRESP a = $W1$ @ ts.cnt.W1.SID. The SID is $\geq W0$, because we know that the seenSet of $I1$'s cache at the time it performs contains a write that is $\geq W0$ (see above in the proof). Then $I1$ brings a seenId down to the CC that is $\geq W0$, meaning the seenId of addr($W1$) is $\geq W0$. Then the RRESP of $I2$ carries this seenId that is $\geq W0$ back to its shim. And, because write ids only get put into the seenSet once they get accepted in order, we know that all messages that were dispatched prior to the seen id of the RRESP will have already accepted. Therefore, we know the write update of $W0$, if one was on the way, will have to have accepted at the shim before the RRESP of $I2$ can be accepted. Therefore, if addr($W0$) is in the shim cache of $I2$, $W0$ (or a newer value) will be in the shim cache. So $I2$'s shim will have seen $W0$.
       Alternatively, $I2$'s shim may have "checked in" to updates from the CC to addr($W0$) only after $W0$ was performed, so a write update of $W0$ will not be on the way. However, if addr($W0$) is in the shim cache by the time $I2$ is performed, this means that a newer value than $W0$ must have been pulled into the shim cache, so our claim still holds.
    2. From shim cache. This case only applies for R_{acq}, since R_{SC}'s always go through the CC. Then $I1$ arrived as a write update and updated the shim cache. The msg was $W1$@ts.cnt.$W1$.SID, for SID $\geq W0$ (see above). If addr($W0$) is not in the shim cache, then by our definition of seen, we already know that $I2$ has seen $W0$. Now consider the addr($W0$) IS in the shim cache. There are two cases, either $I2$'s shim was a sharer before $W0$ was performed, or it became a sharer after. In the first case, again, the write update cannot accept early unless SID is in the seenSet of $I2$'s shim, and SID cannot be in the seenSet if the prior write update of $W0$ is still on the way. This means that $W0$ was accepted and is in the shim cache. Therefore, by our definition of seen, $I2$ has seen $W0$.
       In the other case, $I2$'s shim may have "checked in" to updates from the CC to addr($W0$) only after $W0$ was performed, so a write update of $W0$ will not be on the way. However, if addr($W0$) is in the shim cache by the time $I2$ is performed, this means that a newer value than $W0$ must have been pulled into the shim cache, so our claim still holds.
  - Case 2: $I2$ is F_{SC} (preceded in sb order by a read R). Suppose R is relaxed, otherwise the same reasoning as Case 1 above applies. There are three ways that R could have read from $I1$:
    1. From CC. Then addr($I1$) is not in the shim cache or buffer. The return message is RRESP $W1$@ts.cnt.$W1$.0. This message can accept early no matter what its shim has seen. Suppose it accepts, even though addr($W0$) is in the shim cache, and the update $W0$ has not yet arrived. Then, when the fence performs, it goes to the CC and a FRESP comes back to the shim. This FRESP must accept in order, so we know that the write update to $W0$ must have accepted by the time the FRESP is back. Therefore, we know that $W0$ is in the CC, and that if addr($W0$) is in the shim cache of the fence instruction, $W0$ is in the shim cache by the time the fence completes. Therefore, $I2$ has seen $W0$.
    2. From shim cache. Then $I1$ arrived as a write update and updated the shim cache. We use the exact same reasoning as Case 1.2.2 above.
    3. From the shim buffer. Then $I1$ arrived as a write update and arrived early, being put into the shim buffer before it could be accepted. If addr($W0$) is not in the shim cache, then by our definition of seen, we are done. Now consider that addr($W0$) IS in the shim cache. Then after the read reads from the shim cache, it could be the case

that the write update of $W0$ has not yet arrived, but R has already read $I2$. However, the fence that follows R is the instruction to which the sw edge corresponds, and when the fence is dispatched, it must go through the CC and come back via FREQ and FRESP messages. The FRESP cannot accept early, meaning the write update of $W0$ that was already in the network must accept first before the fence can complete. Therefore, we know that when the fence completes, $W0$ will be in the shim cache, so by our definition of seen, we are done.
Alternatively, $I2$'s shim may have "checked in" to updates from the CC to addr($W0$) only after $W0$ was performed, so a write update of $W0$ will not be on the way. However, if addr($W0$) is in the shim cache by the time $I2$ is performed, this means that a newer value than $W0$ must have been pulled into the shim cache, so our claim still holds.

- Case 2: $I1$ is a F_{SC} (followed in sb order by a write). Now case on $I2$:
  - Case 1: $I2$ is R_{acq} or R_{SC}. There are two cases for how R reads from $I1$:
    1. From the CC. Then the return message will look like: RRESP a = $W1$ @ ts.cnt.$W1$.SID. Notice that $W1$ did not update the seen id since it is a relaxed write, so we do not know how SID relates to $W0$. However, the F_{SC} was processed at the CC before $I1$, meaning that it was sent to $I2$'s shim before the RRESP a = $W1$. And, we know that if a write update of $W0$ was on its way to $I2$'s shim, it would have been sent before the F_{SC}. If the RRESP arrives first before the $W0$, it will have a higher fenceCount than the fenceCount at the shim, because the F_{SC} cannot be processed out of order and therefore would not have arrived yet. This means that the RRESP will be buffered until the fence, and therefore $W0$, have arrived and been processed. So $I2$ will have seen $W0$.
    2. From the shim cache. Then, we know that the write update of $W1$ will be on its way to $I2$'s core. If addr($W0$) is not in I2's shim cache, we are done by our definition of seen, so suppose that it is. Then, a write update of $W0$ would have been sent to $I2$'s shim. The F_{SC} will be sent to all cores, including $I2$'s core, and will be sent between the write updates of $W0$ and $W1$. Again, as in Case 2.1.1, the fence ensures that the write update of $W0$ arrives at the shim before the write update of $W1$, so again, the read will have seen $W0$.
  - Case 2: $I2$ is F_{SC} (preceded in sb order by a read R). Again, assume R is relaxed, otherwise the proofs looks the same as the previous case. There are three cases for how R reads from $W1$:
    1. From the CC. Then the return message will look like: RRESP a = $W1$ @ ts.cnt.$W1$.0. We know that $W0$ hit the CC before $W1$ because all sb-related messages arrive in order. Then, if addr($W0$) is not in S's shim cache, by our definition of seen, we are done. Otherwise, assume addr($W0$) is in the shim cache. Then, $W0$ will have been sent as a write update to S. In the meantime, the read will have arrived at the CC, and RRESP $W1$ @ ts.cnt.$W1$.0 will go back to the shim. This message could accept early before the write update of $W0$ because it is a relaxed message. However, because the following fence would force the write update to be accepted, as explained previously in the proof, $W0$ will be in the shim cache by the time the fence completes. Then the fence will have seen $W0$.
    2. From the shim cache. Then $W1$ arrived as a write update to S. This could have accepted early, because the write is a relaxed write. However, as previously proven, the fence will ensure that if addr($W0$) is in the shim cache and is waiting for an update, that update will arrive before the fence completes, so the fence will have seen W0.
    3. From the shim buffer. This case is clear from the previous cases. The fence ensures that a write update to $W0$ will arrive at the shim before it completes.

Claim (hb writes): Along an hb chain, each synchronizing read reads a write at least as new as the first write in the chain.
Pf. By induction on the number of synchronizing reads in the chain.
BC: hb chain is a single sw edge between instructions $I1$ and $I2$. $I1$ can be either a $W_{rel/SC}$, or a F_{SC} followed immediately by a W_{rlx}. $I2$ can be either a $R_{acq/SC}$, or a R_{rlx} followed by a F_{SC}. In any case, there is only one write in the chain, and it gets read from by the synchronizing read. So the read reads a write at least as new as the first write.
IH: Suppose for an hb chain with n synchronizing reads, each read reads a write at least as new as the first write in the chain.
IS: Consider an hb chain with n+1 synchronizing reads. By the induction hypothesis, the n^{th} synchronizing read read a write at least as new as $W0$. Then, we know that $W0$ must have been in the CC by the time the read happened, by our RF edges claim. And we know that the synchronizing write on this core that synchronizes with the $n+1^{th}$ synchronizing read will hit the CC after the previous read has either dispatched, or read its write from the cache or buffer. This means that the final synchronizing write will hit the CC after the write that the previous read read from, making it newer than $W0$. Therefore, the $n+1^{th}$ synchronizing read will read a write at least as new as $W0$.

Claim (hb;rf extended): Prove that if $W0 \rightarrow_{rf} R0 \rightarrow_{hb} I1$, for $I1$ a write or fence, then $W0$ hit the CC before $I1$.

Pf. By induction.
BC. hb = sb. By the RF edges claim, $R0$ 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 ; sb. This means that $R0$ happens sb-before some instruction $I2$ that synchronizes with $I1$, in this case a fence. By the RF Edges claim, 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, so $I2$ hits the CC after $W0$. We also know that there is some read $R1$ sb-before $I1$ that reads from $I2$ or a sb-future write, per the definition of sw, and by the RF Edges Claim we know that $I2$ 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 $n+1^{th}$ 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 two cases for how $I1$ relates to I_n.

- Case 1: sb. Then we know I_n was processed at the CC before $I1$, so by transitivity, we know that $W0$ hit the CC before $I1$.
- Case 2: sw ; sb. Case on I_n:
  - I_n is a write. This means that I_n was read from by a read sb-before $I1$. By the RF edges claim, 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 the RF Edges claim, 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 $W0$.

Claim (hb;mo extended): Prove that if $I0 \rightarrow_{hb} I1$, for $I0$ and $I1$ writes or fences, then $I0$ hit the CC before $I1$.
Pf. By induction.
BC. hb = sb. Instructions that go through the CC arrive at the CC in sb-order, so our claim holds.
BC. hb = sw. Case on $I0$.

- Case 1: $I0$ is a write, and $I1$ is a fence that is sb-after a read that reads-from $I0$. Then by the RF Edges claim, $I0$ hits the CC before the read is processed, and $I1$ hits the CC after the read is processed due to sb-ordering rules. Therefore, $I1$ hits the CC after $I0$.
- Case 2: $I0$ is a fence that is sb-before a write W, and $I1$ is a fence that is sb-after a read R that reads-from W. Again, by the RF Edges claim, W hits the CC before R is processed. $I0$ hits the CC before W and $I1$ hits the CC after R is processed by sb-ordering rules, so $I0$ hits the CC before $I1$.

BC: hb = sw ; sb. This is different from the previous case because here, we have that $I0 \rightarrow_{sw} R \rightarrow_{sb} I1$, not that $I0$ and $I1$ synchronize directly. Case on $I0$.

- Case 1: $I0$ is a write. This means that $I0$ is read from by a synchronizing read $R2$, which is sb-before $I1$. By the RF Edges claim, $R2$ was processed after $W0$ hit the CC, either because it read $W0$ from the CC or because $W0$ hit the CC and was sent to its shim as a write update. And $I1$ hits the CC after $R2$, or after $R2$ is processed at the shim, so it must hit the CC after $W0$.
- Case 2: $I0$ is a fence. This means that $I0$ is sb-before a write W that is read from by a synchronizing read $R2$, which is sb-before $I1$. By the RF Edges Claim, W hits the CC before $R2$ is processed. And, $I0$ hits before W, and $I1$ hits after $R2$ is processed, so $I1$ hits the CC after $I0$.

IH. Suppose for an hb chain starting with $I0$ a write or fence, and n writes/fences following, each write/fence hits the CC after $W0$.
IS. Suppose there is an hb-chain ending with $I1$, the $n+1^{th}$ 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 two cases for how $I1$ relates to I_n.

- Case 1: sb. Then we know I_n was processed at the CC before $I1$, so by transitivity, we know that $W0$ hit the CC before $I1$.
- Case 2: sw ; sb. Case on I_n:
  - I_n is a write. Then $I1$ is a fence, and there is a read sb-before $I1$ that reads from I_n. By the RF Edges claim, I_n hit the CC before the read performed, and $I1$ hit the CC after the read performed because instructions perform in program order. Therefore, $I1$ hit the CC after I_n, and by the IH and transitivity, $I1$ hit the CC after $W0$.
  - I_n is a fence. This means that I_n is followed by a write in program order, and this write is read from by a read sb-before $I1$, a fence. Again by the RF Edges claim, this write hits the CC before the read performs, meaning that the

fence also hits the CC before the read performs. Since $W1$ is sb-after the read, it hits the CC after the read performs or is dispatched to the CC, meaning $W1$ hits the CC after I_n. By the IH and transitivity, then $I1$ hits the CC after $W0$.

- Case 3: sw ; sb. Case on I_n:
  - I_n is a write. This means that I_n was read from by a read sb-before $I1$. By the RF edges claim, we know that I_n hit the CC before the read that read from it performed, and $W1$ performed after this read, meaning $W1$ hit the CC after $W0$.
  - This means that I_n is followed by a write in program order, and this write is read from by a read sb-before $I1$. Again by the RF Edges claim, this write hits the CC before the read performs, meaning that the fence also hits the CC before the read performs. Since $W1$ is sb-after the read, it hits the CC after the read performs or is dispatched to the CC, meaning $W1$ hits the CC after I_n. By the IH and transitivity, then $I1$ hits the CC after $W0$.

Claim (sb reads): if $R0 \rightarrow_{sb} R1$, then $\forall$ instructions $I1$ s.t. $I1 \rightarrow_{seen} R0$, $I1 \rightarrow_{seen} R1$.
Pf. Suppose there is some write $W0$ s.t. $W0 \rightarrow_{seen} R0$. Then $W0$ hit the CC before any message associated with $R0$, and if addr($W0$) is in the shim cache of $R0$'s shim, then a value at least as new as $W0$ was in the cache.

- Case 1: addr($W0$) was not in the shim cache at either $R0$ or $R1$. Then we are done by our definition of seen.
- Case 2: addr($W0$) was brought into the cache between $R0$ and $R1$ (either because it wasn't in the cache at $R0$, or it was but was then evicted).
  - Case 1: $W0$ is brought into the cache via a write. Then, since $W0$ was seen by $R0$, which happened sb-before the write that brought addr($W0$) back into the cache, this last write must have hit the CC after $W0$. Therefore, it has written newer data to the cache, which cannot be overwritten with stale data. So $R1$ will see a value at least as new as $W0$.
  - Case 2: $W0$ is brought back into the cache via a read. Then, $R1$ cannot perform until all prior instructions perform, so it must wait until the read that reads from addr($W0$) completes. And since this prior read hits the CC after $W0$, it will read a value at least as new as $W0$ and put this value into the shim cache. Therefore, $R1$ will see a value of addr($W0$) that is at least as new as $W0$, meaning it has seen $W0$.
- Case 3: addr($W0$) was evicted after $R0$ and was not brought back into the cache. Then we are done by our definition of seen.
- Case 4: addr($W0$) was never evicted. Then because writes to the shims do not contradict mo, a value at least as new as $W0$ must be in the shim cache. Therefore, $R1$ has seen $W0$.

Claim (hb Write Read): If $W0 \rightarrow_{hb} R0$, then $W0 \rightarrow_{seen} R0$.
Pf. By induction.
BC: hb = sb. Then $W0$ was written into the shim cache, and arrived at the CC before $R0$ (if $R0$ was sent to the CC). If addr($W0$) was evicted before $R0$ performed, then by the definition of seen, we are done. Otherwise, we know that a value at least as new as $W0$ is in the shim cache, since writes to the shims do not contradict mo. So again by our definition of seen, we are done.
BC: hb = sw. Then $R0$ reads from $W0$. By the RF Edges claim, $W0$ must be in the CC before $R0$ arrives. And, when $R0$ reads $W0$, it will put this value in the shim cache (unless a newer value is already there). So by our definition of seen, $R0$ will have seen $W0$.
IH: Suppose for an hb chain involving n reads (where reads that initiate an sw relation going into a following fence are counted), each read has seen $W0$.
IS: Consider an hb-chain of n+1 reads. By the IH, the nth read has seen $W0$, meaning $W0$ was in the CC before any messages associated with the read arrived, and if addr($W0$) was in the shim cache of the nth read, then a value at least as new as $W0$ was in the cache. There are two cases for the relationship between R_n and $R0$:

- sb. This means $R_n \rightarrow_{sw} R0$. By Claim (sb reads), $R0$ has seen everything R_n saw, meaning $R0$ has seen W0.
- sb;sw. This means that for some write $W1$, $R_n \rightarrow_{sb} W1 \rightarrow_{sw} R0$. By the SW Edges 2 claim, we know that any write that happens hb-before an instruction I1 will be seen by an instruction I2 that synchronizes with it. Therefore, since $W0$ happens hb-before W1, it will be seen by $R0$.

Claim (rf;hb Write Read): If $W0 \rightarrow_{rf} R0 \rightarrow_{hb} W1$, for $W1$ a REL or SC write, then any write update or RRESP carrying $W1$ will have seen id at least $W0$.
Pf. By induction on the hb edge.
BC: $W0 \rightarrow_{rf} R0 \rightarrow_{sb} W1$. Case on how $R0$ reads from $W0$.

- Case 1: from the shim cache. Then $W0$ arrived as an update, meaning $W0$ went into the seen set.

- Case 2: from the CC. Then the RRESP of $R0$ carries $W0$ into the seen set of its shim.

Either way, the seen set of $W1$'s shim contains at least $W0$; $W1$ carries the maximum seen id to the CC, where it updates the seen id of its address and shim to be at least $W0$, per the rules of seen id updating. Then, any write update or RRESP that carries $W1$ will carry this seen id, upholding our claim.
IH: For any hb chain with n REL or SC writes, suppose our claim holds.
IS: Consider an hb chain with n+1 REL or SC writes. Case on the relationship between the n^{th} and $n+1^{th}$ writes.

- sb. Then by the induction hypothesis, any update or RRESP of W_n will carry seen id at least $W0$. Both of these messages get their seen id from the seen id stored at the CC of W_n's address and shim. When W_{n+1} arrives at the CC, it updates its seen id to be the max of the current seenPerShim and its seen id. The seenPerShim value will be at least W_n, which will be greater than $W0$, so the seen id of W_{n+1} will be at least $W0$. Then any update or RRESP carrying W_{n+1} will carry seen id at least $W0$.
- sw;sb. Then some read R_n reads from W_n, and R_n is sb-before W_{n+1}. Case on how R_n reads from W_n.
  - Case 1: From the shim cache. Then W_n arrived and was placed into either the seenSetCache or seenSetBuffer. Either way, when W_{n+1} performs, it will carry W_n or higher as its seen id, and will update the CC's seen id of its shim and address to be W_n. Then any update or RRESP carrying W_{n+1} will have seen id at least $W_n > W0$.
  - Case 2: From the CC. Then R_n updates the seenPerShim of its shim to be at least W_n, meaning W_{n+1} later updates the seen id of its shim and address to be at least W_n. Then any future update or RRESP will carry at least $W_n > W0$.

Claim (hb;fr;rf): The following structure is not possible: $R1 \rightarrow_{hb} R0 \rightarrow_{fr} W1 \rightarrow_{rf} R1$.
Pf. Case on the hb edge between $R1$ and $R0$. By the definition of fr, suppose some write $W0$ is mo-before $W1$, and $R0$ reads froms $W0$.
BC: hb = sb. Since $R1$ read from $W1$, $R1$ sees $W1$. By Claim (sb reads), $R0$ then sees $W1$. Thus, if $R0$ gets its value from the CC, $W1$ will already be in the CC, so $R0$ will read $W1$ or newer. If $R0$ reads from the shim cache, again $W1$ or newer will be present, so $R0$ will read $W1$ or newer. This contradicts our assumption that $R0$ read from $W0$.
BC: hb = hb ; sw ; sb. This means that $R1$ happens sb-before some instruction $I2$ that initiates synchronization with some read $I3$, and $I3$ is sb-before $R0$. Case on $I2$ and $I3$.

- Case 1: $I2$ is a write $W2$ and $I3$ is a read $R2$. By Claim (rf;hb Write Read), any write update or RRESP carrying $W2$ will have seen id at least $W1$. This implies that $R2$ will not be able to read from $W2$ early without previously observing an update of $W1$. Therefore, if $R0$ reads from the shim cache, it will read a value at least as new as $W1$; the same is true if it reads from the CC because $W1$ hit the CC before $W2$. Therefore, $R0$ will read $W1$ or newer, contradicting our assumption that it reads $W0$.
- Case 2: $I2$ is a write $W2$ and $I3$ is a fence. By the definition of sw, there must be a read $R3$ sb-before $I3$ that reads from $W2$. Case on how $R0$ reads its value.
  - Case 1: from the shim cache. By Claim (hb;rf), $W1$ hit the CC before $W2$, and by Claim (RF Edges), $W2$ hit the CC before $R3$ either hit the CC or completed locally. Since the fence is sb-after $R3$, then the FREQ hit the CC after $W1$. Therefore, the update of $W1$ will reach the shim cache before the FRESP, meaning $W1$ will be in the shim cache before $R0$ can complete, meaning it will read $W1$. This is a contradiction.
  - Case 2: from the CC. Then, the RRESP of $R0$ will hit the CC after the FREQ, meaning it will hit after $W1$. Thus, it will read a value at least as new as $W1$, a contradiction.
- Case 3: $I2$ is a fence. Then it must be the case that $R2$ read from some write sb-after $I2$, by the definition of sw. Case on how $R2$ reads from this write.
  - Case 1: from the shim cache. Then the write arrived to the CC after the fence, and the FREQ must have already been accepted at the shim before the write, since fences prevent reordering of messages across them. And, by Claim (hb;rf extended), $W1$ hit the CC before the fence. Therefore, if $W1$ was sent as an update to $R2$'s shim, then it will have arrived before the write that $R2$ read from. So it $R0$ reads its value from the shim cache, it will read $W1$ or newer. Otherwise, if it reads from the CC, its RREQ will hit the CC after the fence and therefore after $W1$, meaning it will read $W1$ or newer.
  - Case 2: from the CC. Then by Claim (RF Edges), the RREQ will hit the CC after the write it reads from, and therefore after the fence $I2$. And by Claim (hb;rf extended), $W1$ hit the CC before the fence, meaning the RREQ will read $W1$ or newer.

  In either case, we have a contradiction that $R0$ reads from $W1$ instead of $W0$.

1.2.2 Coherence Axiom.

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

- rf
  If a read read-from a write, then the write hit the CC before the read performed (per the RF edge claim), and then either the read arrived at the CC and read the value, or the write was sent as an update to the reading shim, which later read the value. However, since the write that it reads from is the last instruction in the chain, then by Claim (hb;rf extended) above, this write will hit the CC after the write that the hb-prior read read from. This means that the write will have to have hit the CC before itself, which is not possible. So this case will not happen.
- mo
  By Claim (hb;mo extended) above, we know that all writes in an hb chain hit the CC after the writes that happen before them in the chain. Since the eco edge is mo, the two instructions connected by the hb edge in this case are writes. Therefore, the last write in the hb chain hits the CC after the first write. However, these writes are related in the opposite direction by mo, meaning the second write hit the CC before the first write, which is a contradiction.
- fr
  If a write $W1$ is related to a read $R1$ via fr, then the read read from a previous write $W0$ in mo. By Claim (hb Write Read) above, we know that $R1$ has seen $W1$. However, by the definition of seen, this means that the $R1$ would have read $W1$ or newer, not $W0$, because $W1$ or newer would either be in the shim cache of the reading core, or it would be in the CC for a RRESP to the CC to read. But then we would have a rf relation between these two instructions, not a fr, a contradiction.
- fr ; rf
  We have the following structure: $R1 \rightarrow_{\text{hb}} R0 \rightarrow_{\text{fr}} W1$, and R1 reads from W1. This structure is not observable by Claim (hb;fr;rf).
- mo ; rf
  We have the following structure: $R1 \rightarrow_{\text{hb}} W0 \rightarrow_{\text{mo}} W1$, and R1 reads from W1. By Claim (hb;rf extended) above, $W0$ will hit the CC after the $W1$. However, this contradicts mo order, which implies that $W0$ hit the CC before $W1$. So we have a contradiction.

1.2.3 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 W 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 W 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.

1.2.4 SC Axiom.

Edge Descriptions

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

Claim (SW sink seenId): if $W0 \rightarrow_{\text{hb}} W1$, for an hb chain consisting of only writes and reads (no fences), then the seen id of $W1$ at the CC will be at least $W0$.

Pf. By induction.

BC: hb = sb. Then $W0 \rightarrow_{\text{sb}} W1 \rightarrow_{\text{sw}} R1$. When $W0$ goes to the CC, it sets its shim's seenPerShim value to $W0$. Then when $W1$ arrives, it sets its address's seen id to the maximum of seenPerShim and its message's seen id, which will be at least $W0$.

BC: hb = sw;sb. Then we have $W0 \rightarrow_{\text{sw}} R \rightarrow_{\text{sb}} W1$. Since the hb chain only consists of reads and writes by assumption, we do not consider the case where the sw source and sink instructions are fences. Case on how R reads from $W0$.

- From the CC. Then when R reads from $W0$, it carries $W0$ with its RRESP back to its shim and puts $W0$ in the seenSet (if it accepts on time) or seenSetBuffer (if it accepts early). Either way, when $W1$ then goes to the CC, it carries max(seenSet, seenSetBuffer) to the CC as its message seen id, which is at least $W0$. Then it sets its seen id at the CC to be the max of its message's seen id and the seenPerShim value of its source shim, which will again be at least $W0$.
- From the shim cache. Then a write update of $W0$ arrived to the shim cache. Again, we know that $W0$ set its seen id to be at least $W0$ when it hit the CC, and if it was allowed to accept at the shim, then its seen id was previously accepted in order at the shim. Then, $W0$ will be put into either the seenSet or seenSetBuffer, and using the case reasoning from the previous case, this means that $W1$'s seen id at the CC will be set to be at least $W0$.

IH: Suppose for an hb-chain of n writes, each write's seen id is at least $W0$.

IS: Consider an hb-chain with n+1 writes. Case on how W_{n} relates to $W_{\text{n+1}}$.

- sb. By the same reasoning as in the first base case, we know that when $W_{\text{n+1}}$ arrives to the shim, its seen id is set to be at least W_{n}. And $W_{\text{n}} > W0$ because it arrived after $W0$ to the CC (by Claim (hb;mo extended)), so $W_{\text{n+1}}$'s seen id is at least W0.
- sw ; sb. Therefore, $W_{\text{n}} \rightarrow_{\text{sw}} R \rightarrow_{\text{sb}} W_{\text{n+1}}$. Then, when R reads from W_{n}, it adds W_{n} to the seenSet or seenSetBuffer. Either way, when $W_{\text{n+1}}$ goes to the CC, it sets the seenId to max(seenPerShim, msg.sId). Its msg.sId is the maximum seen id in its shim's two seen sets, which will be at least $W_{\text{n}} > W0$. So our claim holds.

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

Pf. AFSOC W arrives to the CC before I_{n}. Case on how R reads its value.

- Case 1: Suppose R reads from the CC. By Claim (hb write read), we know that R has seen I_{n}. This means that I_{n} was already in the CC before R went to the CC, and since W arrived before I_{n}, then W is already in the CC cache by the time R arrives. Therefore, R will read a value at least as new as W, contradicting the definition of the fr edge between R and W.
- 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 is a sequence of sb edges. We know that a write update of W will be on the way to the shim before I_{n} reaches the CC. Therefore, when I_{n} performs, since it is a W_{SC} or a fence, no other instruction may perform until it receives its acknowledgement back. The W update, having been sent before the acknowledgement, will already have arrived at the reading shim by the time I_{n} completes, and therefore that R will eventually read a value at least as new as W. This would violate the fr edge between R and W.
  - Case 2: hb = sw + sb chain (at least one sw edge). First, realize that if there is some fence $F1$ along this chain, then by Claim (hb;mo extended), I_{n} hit the CC before $F1$, meaning W hit the CC before $F1$. When $F1$ reaches the CC, it will send out FREQ's to every core, and messages to the cores may not get reordered across FREQs. Since the W update was sent before the FREQ, and the write that is read from by R's core is ordered after it, then W will

arrive to R's shim before the sink read on R's core is allowed to complete. Therefore, R would read W or newer, contradicting the fr edge. Therefore, consider an sb-sw chain with only reads and writes, no fences. Then we have: $I_n \rightarrow \text{hb} W0 \rightarrow_{sw} R0 \rightarrow_{sb} R$. Case on the sw edge.

* Case 1: $R0$ is SC and misses at the shim cache. Then it will go through the CC and will be ordered after the W. Since SC instructions arrive in order, $R0$ will not receive its RRESP until after W arrives, meaning R will read a value at least as new as W. This contradicts the fr edge.
* Case 2: $R0$ is a R_{acq} that misses at the shim cache. Then it goes through the CC and reads from $W0$ there. By Claim (hb;mo extended), we know that W hit the CC before $W0$, meaning that the only way that R from-reads W is if W's update arrives out of order with $R0$'s RRESP. By Claim (SW sink seen id), $W0$ will set its seen id at the CC to be at least the seen id of the first write in the chain. When $R0$ reads from the CC, the seen id of its RRESP will be at least the write id of I_n, meaning the only way it can accept early is if a message carrying I_n or newer as a write id accepts in order at the reading shim before it arrives. Since W arrived at the CC before I_n, then we know that $R0$'s RRESP cannot accept in order until the W accepts, meaning R will read from a value at least as new as the W. Contradiction.
* Case 3: $R0$ hits at the shim cache. We then know that $W0$ hit R's shim before $R0$ performs. Since $W0$ will have seen I_n by Claim (hb;mo extended), then I_n will have hit the CC before it, meaning W would have also hit the CC before it. Therefore, the only way to preserve the fr edge between R and W is to have $W0$'s update arrive out of order w.r.t. the write update of W. So, by Claim (SW sink seen id), each synchronizing write's seen id will be at least the seen id of the first write in the chain. And the only way that $W0$ can accept early is if a message carrying its seen id accepts in order at the reading shim before it arrives. Since we know that this message was serialized at the CC after the W (due to the W having a lower seen id than I_n), then we know that $W0$ cannot accept in order until the W accepts, meaning R will read from a value at least as new as the W. Contradiction.

Claim (psc W/F order): writes and fences in psc chains hit the CC in psc order (i.e. if $I1 \rightarrow_{psc^*} I2$, then $I1$ hits the CC before $I2$, for $I1$ and $I2$ writes or fences.
BC: 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.
  - hb. By Claim (hb;mo extended), $I1$ hits the CC before $I2$.
  - 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. If psc_fence is an hb edge, we are done by base case 1.2, so consider F;hb;eco;hb;F. Case on the eco edge.
  - rf. Then we have $F0 \rightarrow_{hb} W \rightarrow_{rf} R \rightarrow_{hb} F1$. By Claim (hb;mo extended), Claim (hb;rf extended), and transitivity, $F0$ hit the CC before $F1$.
  - mo. Then we have $F0 \rightarrow_{hb} W0 \rightarrow_{mo} W1 \rightarrow_{hb} F1$. By Claim (hb;mo extended), the definition of mo, and transitivity, $F0$ hit the CC before $F1$.
  - fr. Then we have $F0 \rightarrow_{hb} R \rightarrow_{fr} W \rightarrow_{hb} F1$. By Claim (hb;fr), Claim (hb;mo extended), and transitivity, $F0$ hits the CC before $F1$.
  - mo;rf. Then we have $F0 \rightarrow_{hb} W0 \rightarrow_{mo} W1 \rightarrow_{rf} R \rightarrow_{hb} F1$. By Claim (hb;mo extended), the definition of mo, Claim (hb;rf extended), and transitivity, $F0$ hit the CC before $F1$.
  - fr;rf. Then we have $F0 \rightarrow_{hb} R0 \rightarrow_{fr} W \rightarrow_{rf} R1 \rightarrow_{hb} F1$. By Claim (hb;fr), Claim (hb;rf extended), and transitivity, $F0$ hit the CC before $F1$.

BC: $I1 \rightarrow_{psc^*} I2$ for a chain of psc reads in between $I1$ and $I2$. WTS if $I1 \rightarrow_{psc^*} I2$ for the instructions in psc* all reads, then $I2$ hit after $I1$. Specifically, we have: $I1 \rightarrow_{psc} R1 \rightarrow \ldots \rightarrow_{psc} Rk \rightarrow_{psc} I2$. The only psc edge that relates two reads is hb, so this simplifies to: $I1 \rightarrow_{psc} R1 \rightarrow_{hb} Rk \rightarrow_{fr} I2$. The first psc can only be hb (that's the only edge relating writes or fences to reads), and the second psc edge can be either hb or fr. If both psc are hb, then we are done by Claim (hb;mo extended). So, consider the case: $I1 \rightarrow_{hb} Rk \rightarrow_{fr} I2$. By Claim (hb;fr), $I2$ hits the CC after $I1$.
IH: Suppose for a psc chain involving n W/F instructions, all W/F instructions hit the CC in psc order.
IS: Consider a chain with n+1 W/F instructions. By the IH, we know all $I1 \ldots I_n$ writes and fences hit the CC in order, so we need only show that In hits the CC before I_{n+1}. Case on how I_n and I_{n+1} are related.

- Case 1: WTS if $I_n \rightarrow_{psc} I_{n+1}$ for no intermediate reads, then I_{n+1} hit after I_n. We have already proven this in the first base case, so we are done.

- Case 2: WTS if $I_n \rightarrow_{psc^*} I_{n+1}$ for psc* made up only of intermediate reads, then I_{n+1} hit after I_n. We have also already proven this in the second base case, so we are done.

Claim (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. Only hb edges can occur between two read instructions. Case on the hb edges' types.

- All hb edges just consist of sb edges. Then we would have an sb cycle, which is not possible because MemGlue executes all instructions in sb order.
- There is at least one sw edge in the hb cycle. Consider the source and sink instructions of some sw edge, $I1$ and $I2$ ($I1 \rightarrow_{sw} I2$). If $I1$ is a write and $I2$ is a read, then $I1 \rightarrow_{rf} I2$, and $I2 \rightarrow_{hb} I1$. This violates the coherence axiom, which we have already proven the MemGlue does not do, so this is not observable in MemGlue. Now suppose $I1$ and $I2$ are fences. Then we have $F1 \rightarrow_{sb} W \rightarrow_{rf} R \rightarrow_{sb} F2$. By the definition of hb, we still have the $R \rightarrow_{hb} W$, and $W \rightarrow_{rf} R$, so the coherence axiom is still violated. The cases where one of $I1$ or $I2$ is a fence follow the same reasoning.

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 psc has a cycle, then the behavior is not observable in MemGlue.
Pf. Consider an arbitrary psc cycle. By Claim (psc reads), this cycle must contain at least one write or fence. Let I1 be the write or fence. Then by Claim (psc W/F order), 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.

1.2.5 No-Thin-Air Axiom.

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

Claim (W;rf;R;sb|rf): 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 $W0$ hit the CC before $W1$.
Pf. By induction.
BC: $W0 \rightarrow_{rf} R0 \rightarrow_{sb} W1$. 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 $W1$ hit the CC after $W0$.
IH: Suppose for a (sb | rf) chain starting with $R0$, and contains n writes, each write hits the CC after $W0$.
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}.

- 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 $W0$.
- rf; sb. Then W_n gets read from by some read R_n, which happens sb-before W_{n+1}. By the RF Edges claim, 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 hits the CC before W_{n+1}.

Therefore, W_{n+1} hits the CC after $W0$.

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

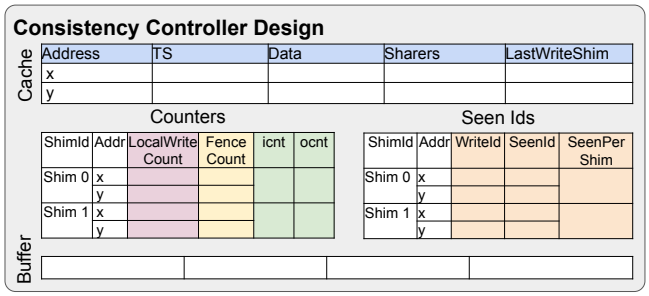
Msg Type	Stren	Addr in buffer	Cache State	Previous read pending	Cache Action	Message to Network			Shim Action
						Src	Msg	Dst	
WRITE A D	RLX REL	-	-	-	Perform write cache[A].TS++	ShimID	WRITE S A D cache[A].TS ocnt 0 max(seenSetCache,seenSetBuffer)	CC	ocnt++
	SC	-	-	-	Perform write cache[A].TS++	ShimID	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.