

System-Centric Specifications for the SC- Memory Model for Java
1/7/2004 3:09 AM

This document describes a high-level system-centric specification, and provides a proof that systems that obey this specification obey the SC- memory model. It is then relatively easy to see that current optimizations are allowed by this specification.¹

Some definitions and notation

Result of an execution: All the reads and the values returned by the reads in the execution.

We use **po** for program order.

Synchronization order (synch), defined on synchronization accesses of an execution.

We require an execution to specify a total order on *conflicting* synchronization accesses such that a read returns the value of the last write in this order. This order is called the synchronization order. It can be naturally derived in systems that have a single point of serialization for conflicting accesses to the same (synchronization) location; e.g., most cache-coherent systems² or systems with no caches. For systems that do not directly provide a serialization point for conflicting synchronization accesses, such an order must be carefully derived so it obeys other constraints imposed on it through the rest of the specification.

Happens-before-synch relation (hbs), defined on synchronization accesses of an execution: I hbs J if I and J are both synchronization memory accesses and (1) I is before J by program order, or (2) I and J conflict and I is before J by synchronization order, or (3) I hbs K hbs J for some K .

Below, we use **relations and graphs** interchangeably. For example, in an hbs graph of an execution, the vertices are memory accesses of the execution and an edge $X \rightarrow Y$ is in the graph if X hbs Y . We say this is a po edge if X po Y or a synch edge if X synch Y in this graph. We say it is a transitive edge if it is neither a po nor a synch edge.

Critical path and critical edge in a graph: This is a path in the graph between two conflicting accesses and that does not have two consecutive program order edges. A program order edge on a critical path is called a critical edge for that graph or relation.

Thus, if $X \rightarrow Y$ is a critical edge for hb, then either X and Y both conflict, or X is a synchronization read, or Y is a synchronization write. If it is a critical edge for hbs, then X and Y are both synchronization. In all cases, X po Y . Further, except for the case where X and Y conflict, both X and Y must be accessed by different threads (and at least one thread other than the initialization thread should write them).

We next define the control (ctl) relation. This relation is conceptually simple, but hard to formalize (and in fact one of the key challenges of this work). Informally, the idea is that a read R controls an access X if R po X and the value returned by R influences whether the instruction instance for X is issued, the address accessed by X , and the value written by X (if X is a write). Further, the read R also controls X if it controls an access Y and $Y \rightarrow X$ could be on a hb or hbs path between two conflicting accesses; i.e., Y po X is a critical edge for either hb or hbs (in other words, read R controls whether an hb or hbs path between two conflicting accesses will occur in the execution or not).

¹ The system-centric specifications described here and the underlying formalisms are analogous to those in Adve's and Gharachorloo's PhD theses. Adve's thesis presents a common set of specifications for a large class of models – the specifications we present here are specialized for the data-race-free-1 model described in the thesis. Several of the concepts here are intuitive, but formalizing them is complex and obscures the exposition. In many places therefore I have used intuitive concepts but pointed to prior work for the formalisms.

² More precisely, we mean that synchronization writes to the same location should be seen in the same order by all processors. Here, we also mean that a write is not made visible to a processor until it is made visible to all processors (referred to as write atomicity in other work). See Adve's and Gharachorloo's theses for a formalization of these concepts.

Below, we define the control relation formally, but in terms of the properties it has to obey. A more constructive, but conservative, form of the control relation can also be defined using the more conventional notion of data and control dependences. For now, we refer the reader to Adve's and Gharachorloo's theses for this definition.³

Control relation (ctl), defined for memory accesses of an execution: We define the control relation for an execution E of program P . We say a read controls an access if it is ordered before that access by the control relation. A control relation for E is correct if $X \text{ ctl } Y$ in E implies that $X \text{ po } Y$ in E and if the following properties hold.

- Consider the set S of reads that control access J in E . Let S' be the set of reads that are in S and also in some execution E_s of program P .⁴ If each read in S' returns the same value in E and E_s , then the following must be true: (1) J occurs in E_s . (2) If $I \rightarrow J$ is a critical edge for hb or hbs in E_s , then I occurs in E .
- If I is a synchronization read and $I \text{ po } J$ in E , then $I \text{ ctl } J$ in E .
- If I controls K in E and if $K \rightarrow J$ is a critical edge for hb or hbs in E , then I controls J in E .

Control+ relation (ctl+), defined on memory accesses of an execution: Control+ is the transitive closure of the control relation and edges of the type $W \rightarrow R$ where R returns the value of W or where $W \text{ synch } R$.

Consistency with a relation: Consider a read that returns the value of a write W in an execution and a relation rel defined on the memory accesses of that execution. Then we say that W is rel -consistent for R if there does not exist a W' such that $W \text{ rel } W' \text{ rel } R$ and if rel does not order R before W . We say that another relation rel' defined on the memory accesses of the same execution is consistent with rel if when $X \text{ rel } Y$, then rel' cannot order Y before X .

System-Centric Specifications

Specification 1: High-level system-centric specification for SC-, also called the hb|hbs|ctl+ consistency condition:

An execution obeys this specification if each read returns the value of a write that is hb-consistent, hbs-consistent, and ctl+ consistent for that read. Further, the hbs relation is acyclic and the ctl+ relation is consistent with the synchronization order relation.

The proof that an execution that obeys specification 1 also obeys SC- follows in the next section. Below, we first describe some more system-centric specifications that obey Specification 1. These specifications are increasingly conservative (and lower-level), but more directly map to current system optimizations and implementation. (The intention is to include all currently implemented system optimizations that I am aware of.)

Specification 2: An execution obeys this specification if for each $X \text{ hb } Y$ or $X \text{ hbs } Y$ or $X \text{ ctl+ } Y$ or $X \text{ ctl+ } \cup \text{ synch } Y$, X completes before Y starts (in real time). This clearly obeys specification 1.⁵

Specification 3: An execution obeys this specification if for each $X \rightarrow Y$ that forms a critical edge for hb, hbs, or ctl+, X completes before Y starts. Further, a synchronization write completes before a read returns its value or before another conflicting synchronization write starts. (Note that for $X \rightarrow Y$ to form a critical edge for ctl+, X must be a read and Y must be a write.) This specification clearly obeys specification 2.

³ Informally, this definition is: consider the transitive closure of conventional control and data dependences, denoted dc . Then a read R controls X if $R \text{ po } X$ and either $R \text{ dc } X$ or there is a Y such that $Y \rightarrow X$ could be on a critical edge of hb or hbs and $R \text{ dc } Y \text{ po } X$.

⁴ I suspect we can restrict E_s to any SC- execution of program P . However, I need to go over the proofs carefully to ensure that works.

⁵ The notion of completion and start is intuitive. For a formal treatment, see the work by Collier and its use in Adve's and Gharachorloo's theses: That work abstracts a system as having a copy of memory for each thread. It breaks a write up into atomic sub-operations, one for each memory copy. A read consists of a single atomic sub-operation that returns the value of the location in its thread's copy. We can say that a write starts when it executes one of its sub-operations and completes when it finishes all its sub-operations. Reads are atomic and start and complete at the same time, since they have only one atomic sub-operation. In most systems, the instant that a write sub-operation or a read sub-operation occurs can be determined in a natural way.

Specifications 2' and 3': We can replace the “X completes before Y starts” with “X is seen before Y by each processor.”⁶ However, we have to add that a synchronization write should complete before a conflicting read returns its value. Again, it is easy to see that specification 2' obeys specification 1 and specification 3' obeys specification 2'.

Theorems and Corollaries

Theorem 1: Consider a program P. For each execution E^* of program P that obeys specification 1, there is an SC-execution with the same result as E^* .

Proof: Follows below.

Corollary 1: A release consistent system obeys SC-.

This is because all executions on a release consistent system obey specification 3.

Corollary 2: A lazy release consistent system obeys SC-.

This is because all executions on a lazy release consistent system obey specification 2'.

Corollary 3: Compilers can reorder two accesses if they are known to not form critical edges.

This follows directly from specification 3 and implies that the standard compiler reorderings are allowed (e.g., two data accesses).

From corollary 3, we can also directly derive that the standard situations for register allocation of accesses and redundant read elimination apply – I will describe these more precisely shortly (these descriptions appear in the thesis).

Some static and dynamic compiler optimizations can exploit the knowledge that some values are always valid to return for a read or that a valid read already returned a certain value. Such optimizations can also be shown to be allowed if the control relation is defined assuming this knowledge. The proof of this is not as direct as the above optimizations and requires knowledge of the proof of theorem 1 below. We also need to formalize these optimizations better, and will do so shortly.

Proof of Theorem 1

To prove theorem 1, we consider an execution E^* of program P that obeys specification 1. We need to show that there is another execution that obeys SC- that has the same result as E^* . Consider an execution E that is the same as E^* , but with a serialization order that is consistent with the hbs relation of E^* . We know that there is such a serialization order since hbs is acyclic and is also consistent with program order. Further, a synchronization read returns the value of the conflicting write ordered last before it in this serialization order (from the definition of synchronization order). We will show that E is an SC- execution. Since E has the same result as E^* , the theorem will be proved.

We note that E also obeys specification 1 since E^* obeys specification 1 and E and E^* have the same accesses and the same hb, hbs, and ctl^+ relations.

To show that E obeys SC-, we need to show that all reads in E can be validated in some total order. The proof proceeds by contradiction. It assumes that at least one read in E cannot be validated. It starts with a maximal set S that consists of all reads that can be validated and such that the set satisfies another property (referred to as property 1 below). It then shows that there is at least one other read R such that R can also be validated and $S \cup R$ also satisfies property 1.

The key to the proof is the specification of property 1 and the choice of the read R as follows.

Let S be the largest set of reads in E that satisfies the following properties, called **Property 1** (S can be empty):

- a) If read R' is in S, then a read ordered before R' by ctl^+ in E is also in S.

⁶ Using the formalism of Collier described in the previous footnote, a processor sees X before Y if the sub-operation of X occurs before the sub-operation of Y in that processor's memory copy, or if one of X or Y does not need a sub-operation in that processor's copy (e.g., if it is a read by another processor).

- b) All reads in S are validated in some total order for E with no other reads validated in between.
- c) There exists an execution, say E_s , such that its serialization order is consistent with its hbs relation, all non-SC reads in E_s are data reads in S , and the following holds for each read R' in S :
 - (i) R' and the write W whose value R' returns in E are also in E_s ,
 - (ii) W is race-consistent for R' in E_s ,
 - (iii) either W and R' form a data race in E and E_s or W hb R' in E and E_s ,
 - (iv) R' returns the value of W in E_s ,
 - (v) if R' is a synchronization read and W is a write that conflicts with R' and occurs in both E_s and E , then either R' hbs W in both E_s and E or W hbs R' in both E and E_s .
 - (vi) if R' is a synchronization read and W_1 and W_2 are writes that conflict with R' and occur in both E_s and E , then either W_1 hbs W_2 in both E_s and E or W_2 hbs W_1 in both E_s and E .

Informally, the idea behind property 1 is to set up an execution E_s that can be used directly to validate another read (R) and to lead us easily to another execution, say E_s' , that satisfies property 1 for S union R . Specifically, part (c)-(i)-(iii) of property 1 are the properties needed to validate a read. Part(c)-(v),(vi) ensure that the order of conflicting synchronization accesses that may impact the value of synchronization reads in the set S is the same in E and E_s .

Our choice of the read R is related to part (a) of property 1 and needs the following lemma.

Lemma 1: For an execution E' that obeys specification 1, the ctl^+ relation is acyclic.

Proof: For a contradiction, assume that the ctl^+ relation has a cycle. Since program order is acyclic, there must be an edge on the cycle of the type $W \rightarrow R$ where W and R conflict and either R returns the value of W or W synch R . The cycle means that R ctl^+ W . If R returns the value of W , then W is not ctl^+ consistent for R , a contradiction. So it must be that W synch R . But then synch and ctl^+ are not consistent with each other, a contradiction. So ctl^+ must be acyclic.

In this proof, we will often consider a property that is assumed to not be obeyed by all reads in E and then identify a read R' such that all reads ordered before R' by the ctl^+ relation in E obey that property, but R' does not obey the property. We refer to such a read as a **first read by ctl^+ in E** [to not obey the property]. By Lemma 1, since the ctl^+ relation is acyclic, such a first read will always exist (there could be many such first by ctl^+ reads in E , since ctl^+ is not a total order).

Going back to the main proof, as mentioned earlier, if the set S contains all reads in E , then the proof is done (since all reads are validated). So there must be at least one read in E that is not in S . We consider a read R in E such that R is a first read by ctl^+ in E to not be in S (i.e., all reads ordered before R by ctl^+ are in S). Then R must occur in E_s since all reads that control it return the same value in E and E_s .

Let $S' = S$ union R . We will show below that S' satisfies property 1, providing a contradiction.

R may be a data or synchronization access. We handle each case separately below.

Case 1: R is a data access.

We first show that R can be validated for E using E_s . We use the following lemma.

Lemma 2: Consider executions E_1 and E_2 of program P such that E_1 obeys specification 1. Let R_1 be a data read that occurs in E_1 and E_2 such that all reads ordered by ctl^+ before R_1 in E_1 occur in E_2 and return the same values in E_1 and E_2 . Let the writes whose value R_1 returns in E_1 and E_2 be W_1 and W_2 respectively. Then W_1 occurs in E_2 and W_1 is hb-consistent for R_1 in E_2 . Further, either W_1 hb R_1 in both E_1 and E_2 or W_1 and R_1 form a data race in both E_1 and E_2 .

Proof: W_1 must be in E_2 . Otherwise, there is a read that controls W_1 , occurs in E_1 and E_2 , and returns a different value in E_1 and E_2 . This read is ordered before R_1 by ctl^+ in E_1 since W_1 is before R_1 by ctl^+ in E_1 . This is a contradiction since R_1 is a first read by ctl^+ in E_1 that returns a different value in E_1 and E_2 .

We next show that $W1$ is hb-consistent for $R1$ in $E2$. The following cases are possible:

Case 1: $R1$ and $W1$ are not related by hb in $E2$.

In this case, $W1$ is hb-consistent for $R1$ in $E2$.

Case 2: $W1$ hb $R1$ in $E2$.

If there is no conflicting write $W2'$ such that $W1$ hb $W2'$ hb $R1$ in $E2$, then again $W1$ is hb-consistent for $R1$ in $E2$. Therefore, assume that there is such a $W2'$.

We show that then it must be that $W1$ hb $W2'$ hb $R1$ in $E1$ as well, but this is a contradiction since $W1$ is hb-consistent for $R1$ in $E1$.

For a contradiction, assume that it is not the case that $W1$ hb $W2'$ hb $R1$ in $E1$. Then some edge on the hb path from $W1$ to $W2'$ and $W2'$ to $R1$ in $E2$ does not occur in the hb relation for $E1$. Let the first such edge going backwards from $R1$ be $X \rightarrow Y$. Either Y is the same as $R1$ or Y ctl+ $R1$.

If $X \rightarrow Y$ is a po edge, then it must be that X is not in $E1$. But then there must be some read that controls X that returns a different value in $E1$ and $E2$, and this read is before Y by ctl+ in $E1$ and so before $R1$ by ctl+ in $E1$, a contradiction.

If $X \rightarrow Y$ is a synch edge, then Y cannot be the same as $R1$ because we know that $R1$ is not a synchronization access. So it must be that Y is a read synchronization ordered before $R1$ by ctl+ in $E1$ and returns the value of a different write in $E1$ and $E2$, a contradiction.

Case 3: $R1$ hb $W1$ in $E2$.

We show (using an argument similar to Case 2) that then it must be that $R1$ hb $W1$ in $E1$ as well, a contradiction since $W1$ is hb-consistent for $R1$ in $E1$. If it is not the case that $R1$ hb $W1$ in $E1$, then some edge on the hb path from $R1$ to $W1$ in $E2$ does not occur in the hb relation for $E1$. Let the first such edge going backwards from $W1$ to $R1$ be $X \rightarrow Y$. Since $W1$ ctl+ $R1$ in $E1$, it follows that Y ctl+ $R1$ in $E1$ as well.

If $X \rightarrow Y$ is a po edge, then it must be that X is not in $E1$. But then there must be some read that controls X that returns a different value in $E1$ and $E2$, and this read is before Y by ctl+ in $E1$ and so before $R1$ by ctl+ in $E1$, a contradiction.

If $X \rightarrow Y$ is a synch edge, then Y is a read synchronization ordered before $R1$ by ctl+ in $E1$ and returns the value of a different write in $E1$ and $E2$, a contradiction.

Thus, it cannot be that $R1$ hb $W1$ in $E2$.

We next show that either $W1$ hb $R1$ in both $E1$ and $E2$ or $W1$ and $R1$ form a data race in both $E1$ and $E2$. There are two cases.

Case 1: $W1$ hb $R1$ in $E2$ but $W1$ and $R1$ form a data race in $E1$.

This case can be argued similar to case 2 above - the po and synch edges constituting the hb path from $W1$ to $R1$ in $E2$ all occur in $E1$, and so $W1$ hb $R1$ in $E1$.

Case 2: $W1$ and $R1$ form a data race in $E2$ but $W1$ hb $R1$ in $E1$.

Then again we can show that the po and synch edges constituting the hb path in $E1$ also occur in $E2$ and so $W1$ hb $R1$ in $E2$.

Going back to the main proof of theorem 1, we note that the chosen read R obeys part (a) of Property 1. By Lemma 2, R also obeys parts c-(i), (ii), and (iii) of Property 1. Thus, R can be validated for E using Es, and so S' also obeys part (b) of property 1.

Further, note that Parts c-(v), (vi) do not apply to R since R is a data access. So if R returns the value of the same write in E and Es, then it obeys all parts of Property 1 for Es, and so S' obeys Property 1, a contradiction.

Therefore, R must not return the value of the same write in E and Es. We next show that even in this case, we can transform Es to another execution, called Es', such that S' obeys property 1 with Es'. We perform the transformation on the serialization order of Es. Note that the serialization order for Es

Informally, the key idea behind this transformation as follows. Let the write whose value R returns in E be W. Then we modify Es to make R return the value of W (W is in Es by Lemma 1). However, this can affect the rest of Es. In fact, it is possible that W may not even be executed in the affected execution, making it impossible to have R return the value of W. Further, our goal is an Es' that obeys property 1, so the impact on Es must be consistent with property 1.

We address the above by exploiting two observations: (1) data reads in S' (i.e., data reads that have been validated) may return non-SC values in Es' (as long as the corresponding write is the same as in E and is hb-consistent for the read in Es'), and (2) if a read R' is in S', then all reads that control it and the reads that control the write whose value R' returns in E are also in S'. If we did not have to worry about synchronization accesses, then with these two observations, it would be relatively easy to show that in Es, we could simply make R return the value of W, leave all other reads in S as they were, and adjust other accesses to ensure uniprocessor correctness and SC values, and the result would be an execution that obeyed property 1 for Es'.

However, with synchronization reads, we have the problem that such a read cannot return a non-SC value. So the mere fact that the correct write exists in the execution does not allow us to force the read to return that value. However, here we can exploit the fact that both E and Es provide relatively strong guarantees for synchronization operations (through specification 1 for E, and by the fact that the serialization order of Es is consistent with hbs of Es). Using these observations, we show how accesses in Es can be reordered so that synchronization accesses in S' also continue to read the value of the writes corresponding to them in E. The key to this are part(c)-(v),(vi) of property 1 – the restructuring seeks to keep the same relative ordering as in E for all conflicting synchronization accesses that determine the value of a synchronization read in S'. This is the most complex aspect of this proof.

Next we give some definitions, then describe the above transformation, and then show that the transformation does indeed lead to the required Es'.

During the transformation of Es', let E' be the transformed serialization order of Es at any point. As will be seen below, some prefix of E' will always be an execution (except perhaps a read may return the value of a write that may occur in the future). The following implicitly applies only to the part of E' that is the longest such prefix.

For the synch relation of E', we use the order implied by the (transformed) serialization order.

The following definition of a required path captures the ordering between conflicting synchronization accesses that must be maintained during the transformation.

Definition: Required path:

Consider a graph consisting of:

- Accesses of E' as the vertices
- Program order edges of E'
- Edges $I \rightarrow J$ such that
 - I and J are conflicting synchronization operations in E'
 - I hbs J in E'
 - If I and J occur in E, then I hbs J in E
 - if either I or J is a read, then the read is in S';

- if J is a read, then I is in E;
- if both I and J are writes, then I and J are in E and there must be some conflicting read in S' that is after these writes in E

We call such edges **ro edges**, for **required order** (since reversing any of these edges could result in inverted synchronization accesses discussed below, which we wish to avoid).

We call a path in the above graph as a required path.

Next we define the notion of inverted synchronization accesses. At some point in the transformation, we will come across synchronization accesses that violate parts c-(v) or (vi) of property 1. We call such accesses as inverted, and need to take special measures to fix them. Informally, these are synchronization accesses for which there is a previous conflicting access in the serialization order, which should not exist (e.g., this is not the order in E, or the previous access is an access that does not occur in E and may make a read return a wrong value). More specifically, we define inverted accesses as follows.

Definition: Inverted synchronization accesses:

A synchronization write W' in E' is inverted in E' if it occurs in E, it conflicts with a read R' in S' , and either:

- (1) W' hbs R' in E but R' hbs W' in E' , or
- (2) there exists a conflicting write W'' in E' and E, and W' hbs W'' hbs R' in E, but W'' hbs W' in E' .

The accesses R' and W'' above are called inverters of W' . There can be more than one inverter for a write.

A synchronization read R' is inverted in E' if it is in S' and either of the following is true:

- (3) There is a conflicting write W' that occurs in E and E' , R' hbs W' in E but W' hbs R' in E' .
- (4) There is a conflicting W' and W'' such that W'' hbs W' hbs R' in E' and R' returns the value of W'' in E.

In both cases, W' is called an inverter of R' .

Transformation of E_s to E_s' .

We transform E_s to E_s' as follows. Start with the serialization order of E_s . Let the prefix before R be the same. Change the value of R to W. Then traverse each execution in the serialization order as follows. Let the current access being traversed be Y.

1. If, based on uniprocessor correctness, Y can no longer be in the execution (because a previous control read returned a different value), delete Y. Call the next access Y and go to 1.
2. If, based on uniprocessor correctness, Y could be in the execution but is not the next access in program order after the last access in the ordering so far, then generate the next access in program order according to uniprocessor semantics. If this access is a shared-memory read, make its value the last conflicting write ordered before the read. Call the next access Y and go to 1.
3. (At this point, Y is the correct next access by program order.) If Y is a data write or a synchronization write that is not inverted, call the next access Y and go to 1.
4. If Y is a read not from S' , make its value the last conflicting write before it in the serialization order. Call the next access Y and go to 1.
5. If Y is a data read from S' or a synchronization access from S' that is not inverted, keep the value of Y the same (this is the same value as in E). Call the next access Y and go to 1.
6. If Y is an inverted synchronization access, let X be the first access in the serialization order E' that is its inverter. Consider all accesses between X and Y that have a required path to Y, but exclude any inverters of Y. Move all of these accesses and Y to before X, maintaining their relative order. Call the first moved access as Y and go to 1.

We next show that the above transformation terminates,⁷ it terminates with an execution, and it terminates with an execution that obeys property 1. But we first need to prove a series of lemmas for this.

Lemma 3: At any invocation of step 6, if I ro J is on a required path to Y in E' and $K \text{ cfl}^+ I$ in E, then K is in S.

⁷ For non-terminating executions, we prove correctness for finite prefixes of the execution. [Need to elaborate on this.]

Proof:

If I is a read, then we know that I is in S' (by definition of ro) and so clearly K is in S (part (a) of property 1).

If I is a write and if J is a read, then J is in S' , I is in E, and $I \text{ hbs } J$ in E. It follows that $I \text{ ctl}^+ J$ in E and so all reads that are before I by ctl^+ in E are also before J by ctl^+ in E. Since J is in S' , it follows that these reads are in S.

If I and J are both writes, then there is a read in S' that is after them by hbs in E. So I is before that read by ctl^+ in E and so all reads that are before I by ctl^+ are in S.

Lemma 4: When step 6 is invoked at Y, then no access before Y in E' is inverted.

Proof: An access can be inverted only due to accesses before it in E' . The first time step 6 is ever invoked on Y, Y is clearly the first inverted access in E' .⁸ The only accesses that are affected by step 6 are those moved during step 6 and those subsequently after them. So accesses before those moved cannot be inverted. We start our traversal from the first access among all affected. Thus, we will catch the first inverted access first. (This can be rephrased more formally as an inductive argument to show that it is true for all invocations of step 6.)

Lemma 5: At any invocation of step 6, there is no required path from an inverter to Y in E' .

Proof: If there is a required path from an inverter to Y in E' , then one such path must be such that there are no consecutive program order edges. Consider such a path.

We will show that all accesses on this path occur in E. But then by definition of a required path, if $I \text{ ro } J$ on this path, then $I \text{ hbs } J$ in E. But this must mean that X (as identified in step 6) is before Y in E. From the definition of an inverted access, $X \text{ hbs } Y$ in E can only be possible for case 4 (in the rest of the cases, if $X \text{ hbs } Y$ in E, then $Y \text{ hbs } X$ in E'). In case 4 of the definition, Y is a read that returns the value of say W'' in E and $W'' \text{ hbs } X \text{ hbs } Y$ in E' . But since X is in E, Y returns the value of W'' in E, and $X \text{ hbs } Y$ in E, it must be that $X \text{ hbs } W'' \text{ hbs } Y$ in E. But then X is inverted with W'' as the inverter, which contradicts Lemma 4.

Thus, to complete this proof, we need only show that all accesses on the above path occur in E.

For a contradiction, consider the last access on the path, say A, that does not occur in E. Then A is not Y and so there is an edge out of A on the above path. The following cases are possible, and we show a contradiction in each.

Case 1. The edge is $A \text{ ro } B$.

By Lemma 3, A is in E.

Case 2. The edge is $A \text{ po } B$ and B is a read.

Then we know that B occurs in E. Either B is Y or has an ro edge out of it on the above path. In either case, B must be in S' . Since A and B are both synchronization, it follows that a read that controls A in E must also control B. All such reads are in S and so have the same value in E' and E and so A must be in E.

Case 3. The edge is $A \text{ po } B$, B is a write, and B is not Y.

Then there is an ro edge out of B. From Lemma 3, a read that controls B is in S. A read that controls A also controls B since both A and B are synchronization, and so such a read is in S. Since such reads return the same values in E and E' , it follows that A must be in E.

Case 4. The edge is $A \text{ po } B$, B is a write, and B is Y.

⁸ We use E' to imply the transformed serialization order. When we use terms such as “first,” “before,” “after,” etc. in the context of E' without specifying another relation, we implicitly mean ordering by the transformed serialization order.

B is an inverted synchronization write, which means that there must be a conflicting read R' in S' such that $B \text{ hbs } R'$ in E . This means that $B \text{ ctl+ } R'$ in E . So all reads that control B are in S and so all reads that control A are in S . Since they return the same values in E and E' , A is in E .

Proof that the transformation terminates.

The only way the transformation may not terminate could be due to step 6, if step 6 gets invoked infinitely often.

Lemma 6: At any invocation of step 6, if Z is an inverted synchronization access, then consider a read R' such that $R' \text{ ctl+ } Z$ in E . Then R' is in S . Further, Z is never deleted or generated through steps 1 and 2 of the transformation.

Proof:

An inverted synchronization access is either in S' or it is a conflicting write before a synchronization read in S' . Thus, all control reads before it by ctl+ in E are in S . From the transformation we note that a read in S never changes its value. Thus, an inverted access is never deleted or regenerated through steps 1 and 2 of the transformation.

Lemma 7: Consider an invocation of step 6 such that $S1$ and $S2$ are two conflicting synchronization operations before the invocation and $S1 \text{ hbs } S2$ in E' , but after the invocation $S2 \text{ hbs } S1$ in the resulting E' . Then $S2$ cannot be an inverter for $S1$ in the resulting E' . That is, the only way an invocation of step 6 can introduce a new inverted access is if the new inverter was generated after the invocation (and did not exist before the invocation).

Proof:

Assume for a contradiction that $S2$ becomes an inverter for $S1$ after the invocation of step 6.

Since $S1$ and $S2$ occurred before the invocation, it must be that $S2$ was moved before $S1$ during the invocation. So it must be that $S2$ was on a required path to the Y chosen for the invocation, but $S1$ was not.

Case 1. $S1$ and $S2$ are both writes.

Then either $S2 \text{ hbs } S1$ in E or there is no conflicting read in S' after them in E . In either case, $S2$ cannot be an inverter.

Case 2. $S1$ is a write, $S2$ is a read.

Then either $S2 \text{ hbs } S1$ in E or $S2$ is not in S' . In either case, $S2$ cannot be an inverter for $S1$.

Case 3. $S1$ is a read, $S2$ is a write.

Then either $S2 \text{ hbs } S1$ in E or $S1$ is not in S' . In either case, $S2$ cannot be an inverter for $S1$.

Notation for k th invocation of step 6: In the following, we will refer to multiple invocations of step 6. To distinguish between X, Y, E' at each invocation, we use a subscript k for the k th invocation (e.g., X_k, Y_k, E'_k , etc.)

Lemma 8: At the end of the k th invocation of step 6, let M_k be the set of accesses that were moved (to before X). Then on the first subsequent traversal of these accesses, step 6 cannot be invoked. That is, the next time step 6 is invoked, it is on an access after M_k and in fact after X_k . Further, if the next invocation of step 6 occurs before any of the accesses after Y_k in E'_k are traversed, then X_{k+1} must be after X_k in E'^{k+1} .

Proof:

Suppose step 6 is invoked on a moved access. Let that access be Y_k and the X corresponding to it be X_k . Then we know from before that Y_k would have been in E'^{k-1} . If X_k was also in E'^{k-1} , then this cannot be. So X_k must be a newly generated access. But then there must be a read R' that controls X_k and returns a different value in E'_k and an access Z such that R' controls X_k po Z ro L where Z ro L was on the required path to Y_{k-1} making Z move (or R was Y_{k-1}). Now X_k and Z are both synchronization. So R' controls Z . But we know that all reads that control an access that is at the head

of an ro edge on required paths to Y_{k-1} are in S (Lemma 3) and all reads that control Y_{k-1} are also in S . So it follows that R' cannot return a different value in E^k , a contradiction.

We next show that step 6 cannot be invoked on X_{k-1} either, on the first traversal subsequent to the k th invocation of step 6. From the above argument, we cannot have a new synchronization access in the moved accesses. But by Lemma 7, X_{k-1} cannot get inverted because of an old synchronization access.

Next we show that if the $k+1^{\text{st}}$ invocation of step 6 is on an access that was before Y_k in E^k , then X_{k+1} must be after X_k in E^{k+1} . This argument is similar to the above.

Consider call k to step 6, with X_k, Y_k . We show that eventually we will come to a point where we traverse an access after Y_k . Thus, we have to make forward progress and will eventually terminate (once we traverse the last (dummy) access).

There are a bounded number of calls to step 6 before we traverse an access after Y_k . Consider all these calls. By Lemma 8, if Y_{k+1} is before Y_k , then X_{k+1} and Y_{k+1} must lie in between X_k and Y_k , as ordered by E^k . In E^{k+1} , Y_k is before X_k and before X_{k+1} . Thus, once an access is chosen for a Y , it cannot be chosen for a Y again for any of these calls. Since no new Y 's can be generated in this process, there are a limited number of calls possible.

The transformation terminates with an execution.

To show that the transformed order is a serialization order for some execution, we have to show that the accesses satisfy uniprocessor correctness and a read returns the value of some conflicting write in the execution.

The only steps in the transformation that can affect the above are steps 5 and 6. However, step 6 does not by itself cause any problem because it does not violate program order (Lemma 5) and the values returned by reads are fixed later.

All we need to show is that in step 5, when a read returns the value of a write, that write is in the resulting execution. The reads affected by this step are all from S' . Therefore, the control reads for the writes whose value these reads return are all also in S' . So if all reads in S' return the same values as before, all the writes whose values they read will also be in the resulting execution.

The transformation terminates with an execution that obeys Property 1.

This follows directly from the transformation.

Case 2: R is a synchronization access.

If R is a synchronization access, then choose an E_s that satisfies property 1 for S , but also has the minimal number of inverted – inverter synchronization access pairs.

If R returns the same value in E and E_s , then it must be that there is a write before it in E_s such that it has a different order with respect to R or with respect to another write before R in E_s . (Otherwise, S' obeys property 1.) Call the first such inverted access as Y and apply step 6 of the previous transformation to it, following up with step 1 etc. At the end we have an execution E_s' that has all the properties of E_s with no inverted pair for S' .

If R returns a different value in E and E_s , then again there must be an inverted pair and we can apply the same transformation to the first inverted access.