Analysis of the ECMAScript Memory Model : A Program Transformation Perspective

Akshay Gopalakrishnan

School of Computer Science

McGill University

August 15, 2020

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Computer Science
©2020 Author

Abstract

Abrégé

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Chapter 2

Background

Chapter 3

Related Work

Chapter 4

The Memory Model

4.1 Agents, Events and their Types

Agents Agents represent threads in a concurrent program. As per the standard, they have more meaning than what we refer to here. However, with respect to the memory consistency model, we can safely abstract them to just represent threads/processes.

Agent Cluster Collection of agents concurrently communicating with each other through means of shared memory form an agent cluster. There can be multiple agent clusters. However, an agent can only belong to one agent cluster. Agents communicating through message passing do not belong in the same agent cluster.

For our purpose, we assume just one agent cluster having one shared memory.

Agent Event List (*ael*) Every agent is mapped to a list of events. The list represents the order in which the events are evaluated operationally[1]. We define *ael* as a mapping of each agent to a list of events.

4.2 Events

Agent execution is modelled in terms of events. An event is either an operation that involves (shared) memory access or that constrains the order of execution of multiple events.

[1]The standard refers this to be an Event List, but we find it a bit misleading as it does not signify a list for each agent. Hence we name it as Agent Event List.

4.2.1 Event Types

Given an agent cluster, an *event set* $\boldsymbol{E}$ is a collection of all events from the agent event lists. This set is composed of mainly two distinct subsets as follows:

- **Shared Memory ($\boldsymbol{SM}$) Events**

  This set is composed of two sets of events; those that write to shared memory called Write events ($\boldsymbol{W}$) and those that read from shared memory called Read events ($\boldsymbol{R}$). Events that belong to both Write and Read events are called Read-Modify-Write.

- **Synchronize ($\boldsymbol{S}$) Events** These events only restrict the ordering of execution of events by agents. For instance *lock* and *unlock* type of events can be categorized under Synchronize events. However, this is not stated in the specification[2].

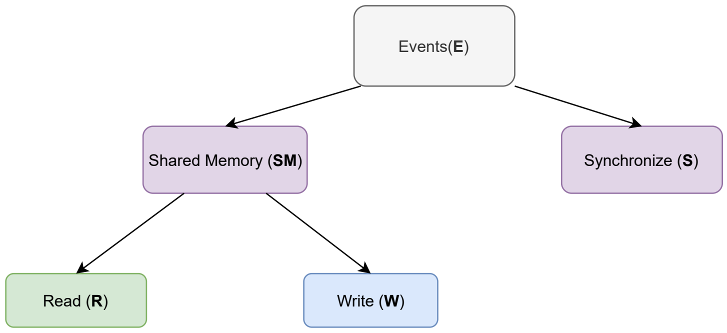


Figure 4.1: The hierarchical categories of different sets of events.

4.2.2 Range ($\Re$)

Each of the *shared memory events* are associated with a contiguous range of memory on which it operates. Range is afunction that maps a shared memory event to the range[3]it operates on. This we represent as a starting index i and a size$s$. So we could represent the range of a write event w as

[2]The features of *Lock* and *Unlock* events is actually not something given to the programmer to use in Javascript. They areused to implement the feature *wait* and $notify$ that the programmer can use which adhere to the semantics of $futexes$ inLinux. Hence, in the original standard of the model, the distinction between lock and unlock is not made, and it is simplystated as Synchronize Event.

$$\Re(w) = (i, s)$$

We define the two binary operators below on ranges:

1. Intersection ($\cap_{\Re}$) - Set of byte indices common to both ranges.
2. Union ($\cup_{\Re}$) - A unique set of byte indices that exist in both the ranges.

Two Ranges can be *disjoint*, *overlapping* or *equal*. We use the binary operators to define these threepossibilities between ranges of events e and d :

1. Disjoint $\Re(e) \cap_{\Re} \Re(d) = \phi$
2. Overlapping $(\Re(e) \cap_{\Re} \Re(d) \neq \phi) \wedge (\Re(e) \cap_{\Re} \Re(d) \neq \Re(e) \cup_{\Re} \Re(d))$ -
3. Equal $\Re(e) \cap_{\Re} \Re(d) = \Re(e) \cup_{\Re} \Re(d)$ - In simple terms, we define equality as $\Re(e) = \Re(d)$

4.2.3 Event Order / Event Access Mode

Order signifies the sequence in which event actions are visible to different agents as well as the order in which they are executed by the agents themselves. In our context, there are mainly three types (in C11 memory model, they are called access modes) for each shared memory event that tells us the kind of ordering that it enforces.

1. **Sequentially Consistent (*sc*)** - Events of this type are *atomic*[4]in nature. There is a strict global total ordering of such events which is agreed upon by all agents in the agent cluster.
2. **Unordered (*uo*)** - Events of this type are considered *non-atomic* and can occur in different orders for each concurrent process. There is no fixed global order respected by agents for such events.
3. **Initialize (*init*)** - Events of this type are used to initialize the values in memory before they are accessed by agent events.

[3]The range as per the ECMAScript standard denotes only the set of contiguous byte indices. The starting byte indexis kept separate. We find this to be unnecessary. Hence we define range to have starting index and size.

All events of type *init* are writes and all Read-Modify-Write events are of type *sc*. We represent the type of events in the memory consistency rules in the format "*event* : *type*". When representing events in examples, the type would be represented as a subscript: $event_{type}$.

4.2.4 Tear Free (tf) or Tearing $!tf$)

Additionally, each shared-memory event is also associated with whether they are tear-free or not. OEvents that tear are non-aligned accesses requiring more than one memory access. Events that are tear-free are aligned and should appear to be serviced in one memory fetch[5].

We represent the tearing of events in the memory consistency rules in the format "*event* : *tf/!tf*". When representing events in examples, the type would be represented as a subscript: $event_{tf/!tf}$.

4.3 Relation among events

We now describe a set of binary relations between events. These relations help us describe the consistency rules.

4.3.1 Read-Write event relations

There are two basic relations that assist us in reasoning about read and write events.

Read-Bytes-From ($\xrightarrow[rbf]{}$) This relation maps every read event to a list of tuples consisting of write event and their corresponding byte index that is read. For instance, consider a read event $r[i...(i+3)]$ and corresponding write events $w_1[i...(i+3)]$, $w_2[i...(i+4)]$. One possible $\xrightarrow[rbf]{}$ relation could be represented as

$$e \xrightarrow[rbf]{} \{(d1, i), (d2, i+1), (d2, i+2)\}$$

[4]The word *atomic* does not imply the events are evaluated using just one instruction. For example, a 64-bit sequentially consistent write on a 32-bit system has to be done with two subsequent memory actions. But its intermediate state of write must not be seen by any other agent. In an abstract sense, this event must appear '*atomic*'.The *atomic* here also refers to implications of whether an event's consequence is visible to all other agents in the same global total order or not. The compiler must ensure that for each specific target hardware, such guarantees are satisfied.

[5]It is not clear whether the alignment is with respect to specific hardware or not. The notion of one memory fetch may not be possible for all hardware practically, but it is something that must appear so. We will see a rule for ensuring this in the memory consistency rules.

or having individual binary relation with each write-index pair as

$$e \xrightarrow[rbf]{} (d1, i),\ e \xrightarrow[rbf]{} (d2, i+1) \text{ and } e \xrightarrow[rbf]{} (d2, i+2).$$

Reads-From ($\xrightarrow[rf]{}$) This relation, is similar to the above relation, except that the byte index details are not involved in the composite list. So for the above example, the *rf* relation would be represented either as $e \xrightarrow[rf]{} (d1, d2)$ or individual binary read-write relation as $e \xrightarrow[rf]{} d1$ and $e \xrightarrow[rf]{} d2$. Figure below is an example of a program with its outcome (read values) shown in terms of reads-from relations.

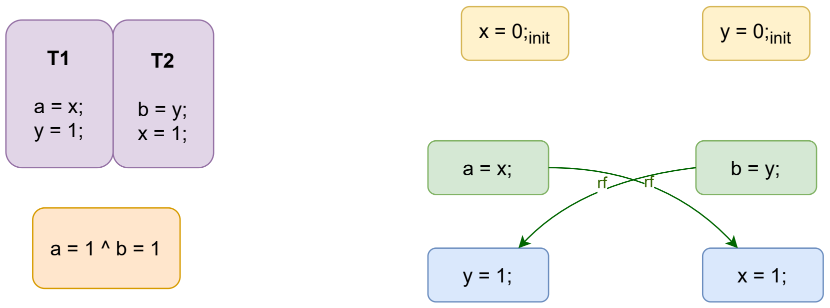


Figure 4.2: An example to show the reads-from relations that are drawn for the example program between read and write events.

4.3.2 Agent-Synchronizes With (*ASW*)

It is a list for each agent that consist of ordered tuples of synchronize events. These tuples specify ordering constraints among agents at different points of execution. So such a list for an agent k would be represented like:

$$ASW_k = \{\langle s_1, s_2\rangle, \langle s_3, s_4\rangle ...\}$$

For every pair in the list, the second event belongs to the parent agent and the first belongs to another agent it synchronized with[6].

$$\forall i, j > 0,\ \langle s_1, s_2\rangle \in ASW_j \Rightarrow s_2 \in ael(k)$$

The figure below shows an example of this relation among two agents.

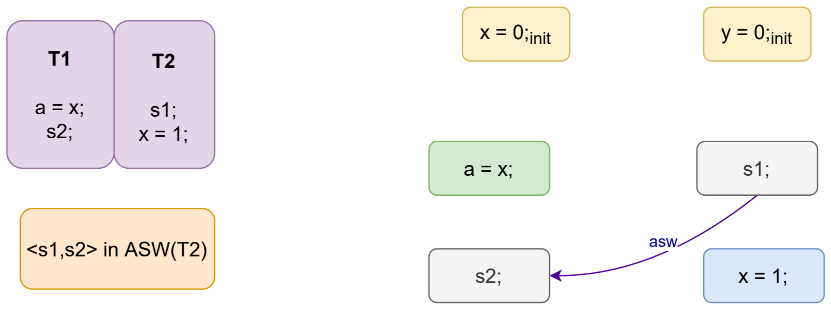


Figure 4.3: An example to show the reads-from relations that are drawn for the example program between read and write events.

4.4 Ordering Relations among Events

4.4.1 Agent Order ($\xrightarrow[ao]{}$)

It is a union of total order among events belonging to the same agent event list. It is analogous to intra-thread ordering. For example, iftwo events e and d belong to the same agent event list , then either $e \xrightarrow[ao]{} d$ or $d \xrightarrow[ao]{} e$.

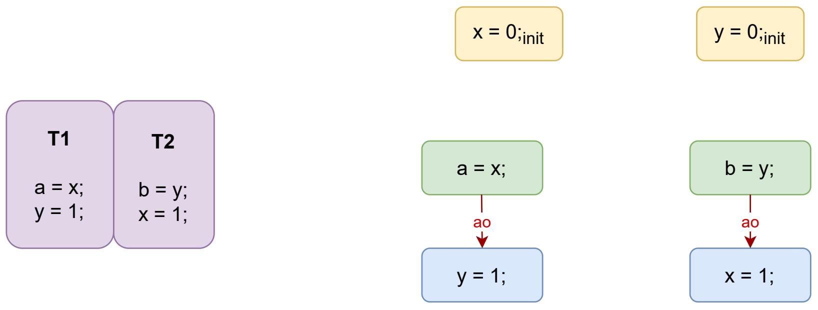


Figure 4.4: An example with agent order among the events.

4.4.2 Synchronize-With Order ($\xrightarrow[sw]{}$)

Binary relation between two events that establish synchronization between multiple agents. It is a composition of two sets:

[6]This is analogous to the property that every unlock must be paired with a subsequent lock, which enforces the condition that a lock can be acquired only when it has been released.

1. All pairs belonging to ASW of every agent belongs to this ordering relation.

$$\forall i,j > 0,\ \langle e_i, e_j \rangle \in ASW \Rightarrow e_i \xrightarrow[sw]{} e_j$$

2. Specific reads-from pairs also belong to this ordering relation[7].

$$(r \xrightarrow[rf]{} w) \ \wedge \ r\!:\!sc \ \wedge \ w\!:\!sc \ \wedge \ (\Re(r)\!=\!\Re(w)) \ \Rightarrow \ (w \xrightarrow[sw]{} r)$$

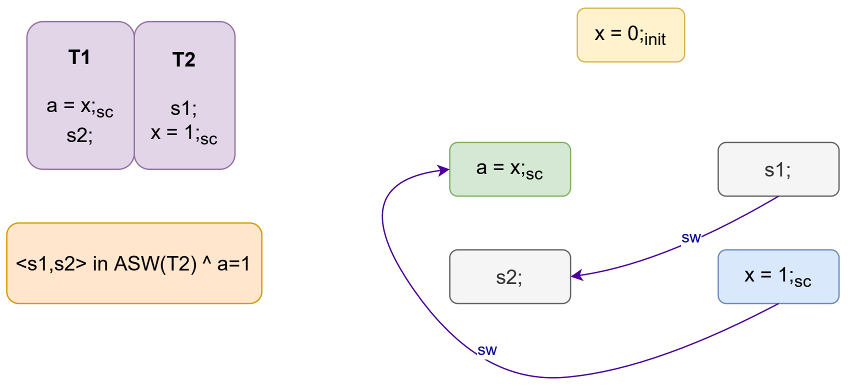


Figure 4.5: An example with synchronize with relations among the events.

4.4.3 Happens Before Order ($\xrightarrow[hb]{}$)

A transitive order on events, composed of the following:

1. Every agent-ordered relation is also a happens-before relation

$$(e \xrightarrow[ao]{} d) \ \Rightarrow \ (e \xrightarrow[hb]{} d)$$

2. Every synchronize-with relation is also a happens-before relation

$$(e \xrightarrow[sw]{} d) \ \Rightarrow \ (e \xrightarrow[hb]{} d)$$

[7]Note that for the second condition, both ranges of events have to be equal. This however, does not mean thatthe read cannot read from multiple write events. (the read-from relation here is not functional.)

3. Initialize type of events happen before all shared memory events that have overlapping ranges with them.

$$\forall e, d \in SM \ \wedge \ e\!:\!init \ \wedge \ (\Re(e) \cap \Re(d) \neq \phi) \ \Rightarrow \ e \xrightarrow[hb]{} d$$

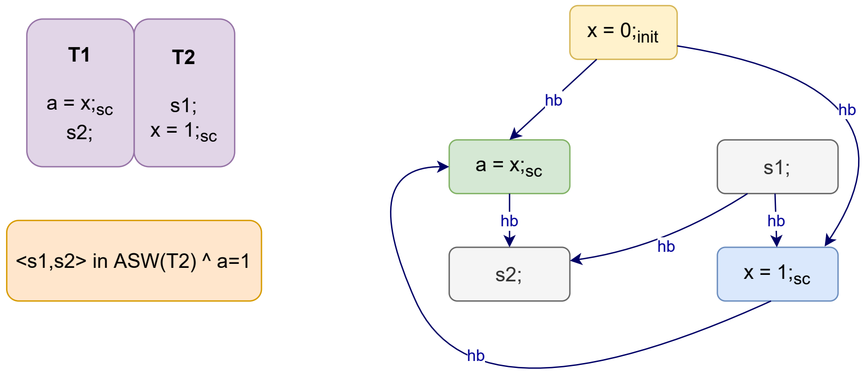


Figure 4.6: An example with all the types of happens-before relations between events.

4.4.4 Memory Order ($\xrightarrow[mo]{}$)

A *total order* on all events that respects happens-before order.

$$e \xrightarrow[hb]{} d \Rightarrow e \xrightarrow[mo]{} d$$

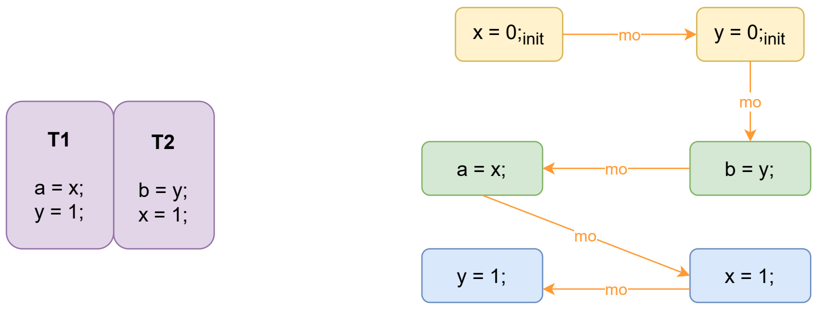


Figure 4.7: An example with a memory order (total) among all events.

An interesting part is that memory order, though total, is a bit undefined as to how it weaves together this total order given different init events. One can certainly make init events weaved among events that occur in each agent, thus making it sort of subjective as to when certain memory fragments are initialized. Discuss with Clark.

4.5 Preliminaries

Before we go into the consistency rules. we define certain preliminary definitions that create a separation based on a program, the axiomatic events and the various ordering relations defined above. This will help us understand where the consistency rules actually apply.

Definition 1. *Program A* program *is the source code without abstraction to a set of events and ordering relations. In our context, it is the original Javascript program.*

Definition 2. *Candidate A a collection of abstracted set of shared memory events of a program involved in one possible execution, with the $\xrightarrow[ao]{}$ relations. We can think of this as each thread having a set of shared memory events to run in a given intra-thread ordering. An example of a candidate is shown in figure 4.8.*

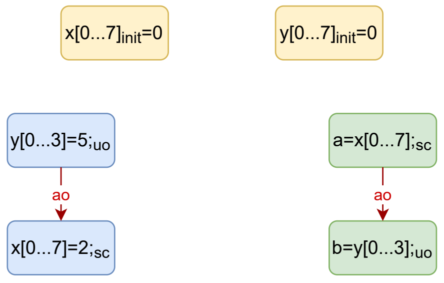


Figure 4.8: An example of a Candidate

Definition 3. *Candidate Execution A Candidate with the addition of $\xrightarrow[sw]{}$, $\xrightarrow[hb]{}$ and $\xrightarrow[mo]{}$ relations. This can be viewed as the witness/justification of an actual execution of a Program. Note that there can be many Candidate Executions for a given Candidate. The following figure shows an example of a candidate execution.*

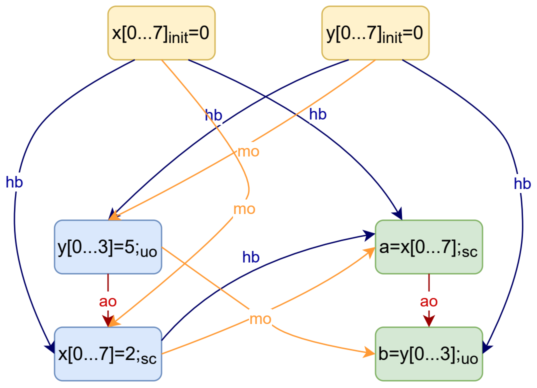


Figure 4.9: An example of an Execution based on Candidate above

Definition 4. *Observable Behavior*

The set of pairwise $\xrightarrow[rf]{}/\xrightarrow[rbf]{}$ *relations that result in one execution of the program. Think of this as our outcome of a program execution.*

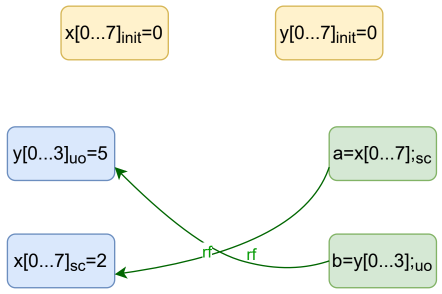


Figure 4.10: Observable Behavior

Definition 5. *Obs We define* Obs_P, Obs_C, Obs_{CE} *as functions that take a program, candidate and candidate execution respectively and give the set of observable behaviors possible by them. We are not concerned with the specific elements in this set, but the relation between the output of each of these functions among each other.*

Consider a program P *whose candidates are* $C_1, C_2, ..., C_n$*. Consider for each candidate* C_i*, the candidate executions* $CE_1, CE_2, ...CE_m$*. Then, we have the following properties that hold:*

$$Obs_P(P) = \cup_{i=1}^{n} Obs_C(C_i)$$
$$Obs_C(C_i) = \cup_{j=1}^{m} Obs_C(CE_j)$$

Consider whether it is required to define Obs_O, since it is a redundant defintion, as it maps every observablke behavior to a singleton set with that observable behavior.

The last definition would call for quite some edits in the proofs ahead. Need to spend separate dedicated time to get them done. Plan this carefully and only after adding all necessary figures where required for each part of the proof.

4.6 Valid Execution Rules (the Axioms)

We now state the memory consistency rules. The rules are on *Candidate Executions* which will place constraints on the possible *Observable behaviors* that may result from it.

4.6.1 Coherent Reads

There are certain restrictions of what a read event cannot see at different points of execution based on $\xrightarrow[hb]{}$ relation with write events.

Consider a read event e and a write event d having at least overlapping ranges:

$$e \in R \;\wedge\; d \in W \;\wedge\; (\Re(e) \cap_{\Re} \Re(d) \neq \phi).$$

- A read value cannot come from a write that has happened after it

$$e \xrightarrow[hb]{} d \;\Rightarrow\; \neg\, e \xrightarrow[rf]{} d.$$

The figure below pictorially depicts the pattern above hwere e cannot read from d.

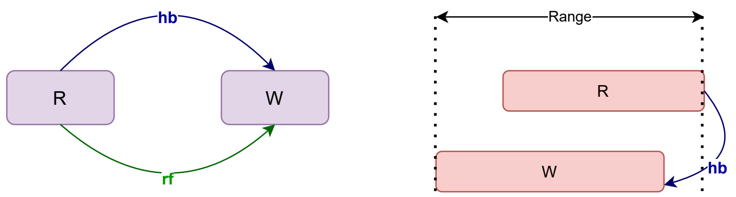


Figure 4.11: A read value cannot come from a write that has happened after it

- A read cannot read a specific byte address value from write if there is a write g that happens between them which modifies the exact byte address. Note that this rule would be on the rbf relation among two events.

$$d \xrightarrow[hb]{} e \;\wedge\; d \xrightarrow[hb]{} g \;\wedge\; g \xrightarrow[hb]{} e \;\Rightarrow\; \forall x \in (\Re(d) \cap_{\Re} \Re(g) \cap_{\Re} \Re(e)),\; \neg\, e \xrightarrow[rbf]{} (d, x).$$

The figure below pictorially depicts the pattern where e cannot read certain bytes from d.

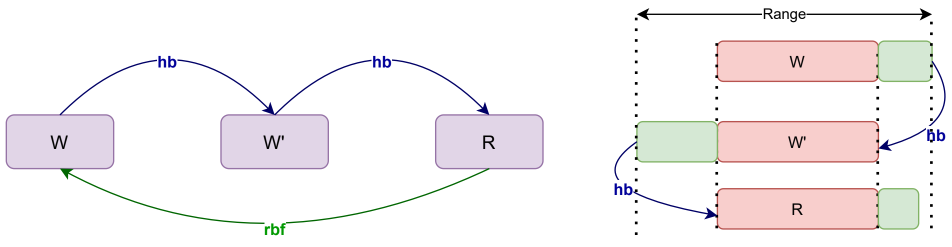


Figure 4.12: A read value cannot come from a write if there is a write that happens between them, writing to the same memory:

4.6.2 Tear-Free Reads

If two tear free writes d and g and a tear free read e all with equal ranges exist, then e can read only from one of them[8].

$$d{:}tf \ \wedge \ g{:}tf \ \wedge \ e{:}tf \ \wedge \ (\Re(d) = \Re(g) = \Re(e)) \ \Rightarrow$$
$$((e \xrightarrow[rf]{} d) \ \wedge \ (\neg\, e \xrightarrow[rf]{} g)) \ \vee \ ((e \xrightarrow[rf]{} g) \ \wedge \ (\neg\, e \xrightarrow[rf]{} d)).$$

The following figure shows the pattern that is disallowed among all tear-free events.

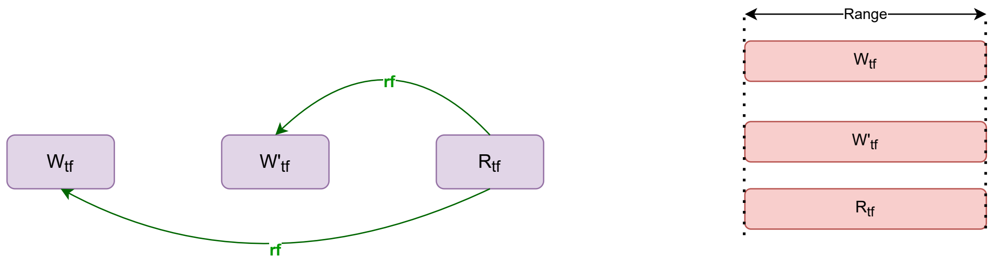


Figure 4.13: Pattern of Tear-free reads

[8]To recap a tear-free event cannot be separated into multiple small events that do the same operation. However, considering different hardware architectures, the notion of tear-free need not necessarily mean this. (eg: A 64bit tear-free write to be done in a 32bit system). In a more abstract sense, we need an event to appear 'tear-free'.

4.6.3 Sequentially Consistent Atomics

To specifically define how events that are sequentially consistent affects what values a read cannot see, we assume the following memory order among writes d and g and a read e to be the premise for all the rules:

$$d \xrightarrow[mo]{} g \xrightarrow[mo]{} e.$$

- If all three events are of type sc with equal ranges, then e cannot read from d

$$d\!:\!sc \;\wedge\; g\!:\!sc \;\wedge\; e\!:\!sc \;\wedge\; (\Re(d)\!=\!\Re(g)\!=\!\Re(e)) \;\Rightarrow\; \neg\, e \xrightarrow[rf]{} d.$$

  The figure below depicts pictorially the pattern that is not allowed by this rule.

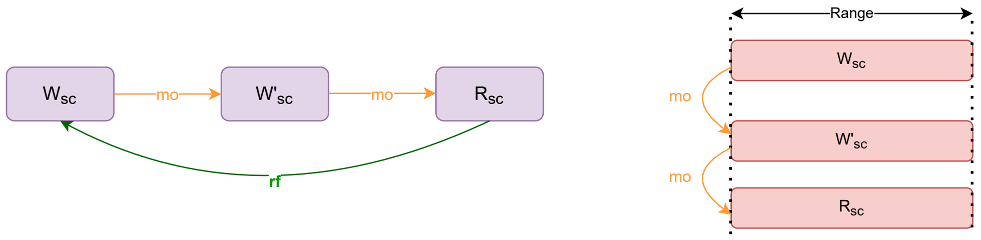


Figure 4.14: A read value cannot come from a write, if there exists a write memory ordered between them and all 3 events are sequentially consistent with equal ranges.

- If both writes are of type sc having equal ranges and the read is bound to happen after them, then e cannot read from d

$$d\!:\!sc \;\wedge\; g\!:\!sc \;\wedge\; (\Re(d)\!=\!\Re(g)) \;\wedge\; d \xrightarrow[hb]{} e \;\wedge\; g \xrightarrow[hb]{} e \;\Rightarrow\; \neg\, e \xrightarrow[rf]{} d.$$

  The figure below depicts pictorially the pattern that is not allowed by this rule.

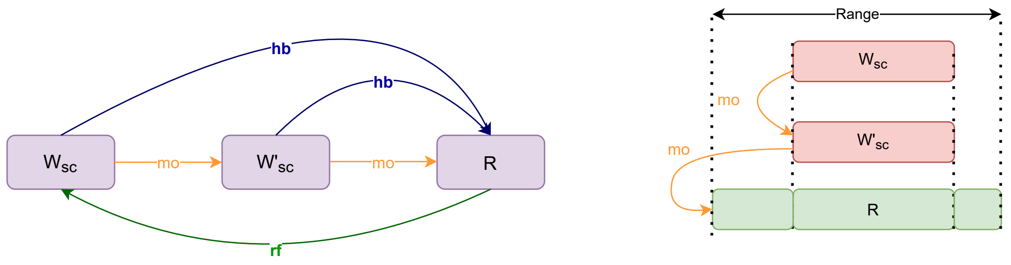


Figure 4.15: A read value cannot come from a write, if there exists a write memory ordered between them and both writes are sequentially consistent with equal ranges.

- If g and e are sequentially consistent, having equal ranges, and d is bound to happen before them, then e cannot read from d

$$g:sc \;\wedge\; e:sc \;\wedge\; (\Re(g)=\Re(e)) \;\wedge\; d \xrightarrow[hb]{} g \;\wedge\; d \xrightarrow[hb]{} e \;\Rightarrow\; \neg\, e \xrightarrow[rf]{} d.$$

The figure below depicts pictorially the pattern that is not allowed by this rule.

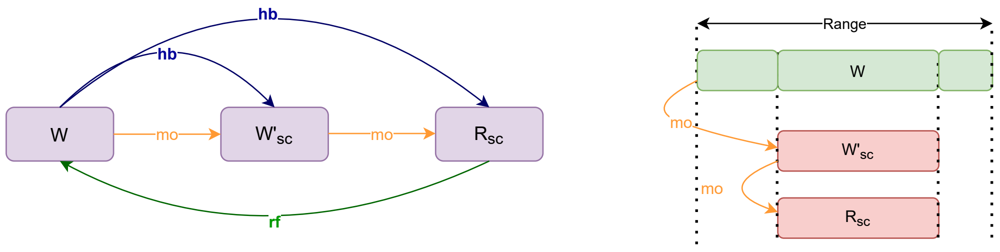


Figure 4.16: A read value cannot come from a write, if there exists a write memory ordered between them and both this write and the read are sequentially consistent with equal ranges.

4.7 Race

4.7.1 Race Condition *RC*

We define ***RC*** as the set of all pairs of events that are in a race. Two events e and d are in a race condition when they are shared memory events:

$$(e \in SM) \;\wedge\; (d \in SM).$$

having overlapping ranges, not having a $\xrightarrow[hb]{}$ relation with each other, and which are either two writes or the two events are involved in a $\xrightarrow[rf]{}$ relation with each other. This can be stated concisely as,

$$\neg\,(e \xrightarrow[hb]{} d) \;\wedge\; \neg\,(d \xrightarrow[hb]{} e) \;\wedge\; ((e,d\in W \;\wedge\; (\Re(d)\cap_{\Re}\Re(e) \neq \phi)) \;\vee\; (d \xrightarrow[rf]{} e) \;\vee\; (e \xrightarrow[rf]{} d)).$$

4.7.2 Data Race *DR*

We define ***DR*** as the set of all pairs of events that are in a data-race. Two events are in a data race when they are already in a race condition and when the two events are not both of type sc, or they have overlapping ranges. This is concisely stated as:

$$e,d\in RC \;\wedge\; ((\neg e:sc \;\vee\; \neg d:sc) \;\vee\; (\Re(e)\cap_{\Re}\Re(d) \neq \Re(e)\cup_{\Re}\Re(d)))$$

The definition for data race also implies that sequentially consistent events with overlapping ranges are also in a data race. This may be counter-intuitive in the sense that all agents observe the same order in which these events happen.

Data-Race-Free (DRF) Programs An execution is considered data-race free if none of the above conditions for data-races occur among events. A program is data-race free if all its executions are data race free. *The memory model guarantees Sequential Consistency for all data-race free programs (SC-DRF).*

4.8 Consistent Executions (Valid Observables)

A valid observable behaviour is when[9]:

1. No $\xrightarrow[rf]{}$ relation violates the above memory consistency rules.
2. $\xrightarrow[hb]{}$ is a strict partial order.

The memory model guarantees that every program must have at least one valid observable behaviour.

[9]There is also some conditions on host-specific events (which we mentioned is not of our main concern) and what is called a chosen read, which is nothing but the reads that the underlying hardware memory model allows. Since we are not concerned with the memory models of different hardware, this restriction on reads is not of our concern.

Chapter 5

Instruction Reordering

5.1 Introduction

Instruction reordering is a common operation in compiler optimization, essential to instruction scheduling of course, but also implicit in loop invariant removal, partial redundancy elimination, and other optimizations that may move instructions. However, whether we can do such reordering freely given a concurrent program using relaxed memory accesses is a bit unclear.

Simple reordering is not straightforward under shared memory semantics The main reason is that memory accesses here, do not just perform the desired operation (i.e Read / Write) but also imply certain visibility guarantees across all the other threads. In our observation, we find that, the relaxed memory model of Javascript prescribe semantics for visibility using the $\xrightarrow[hb]{}$ relations.

Show an example or multiple examples here that enforces visibility due to having sequentially consistent events involved in a Candidate Execution.

What can be done? An example-based analysis exposes to us the problems that might exist when we perform such reordering of events. However, such an analysis, though would work for small programs to identify the possible conditions under which reordering can be done, become infeasible as the programs scale in length and complexity. This is because of the exponential increase in possible executions as the number of threads and program size in general increase. Hence, generalizations by using a small sample size is not something we can afford especially when we want to ensure these program trasnformations are done by the compiler in contrast to being done manually.

Our approach Our solution to this is to construct a proof on Candidate Executions of the original program and the transformed one which exposes the possible observable behaviors it can have. The crux of the proof is to guarantee that reordering does not bring any new $\xrightarrow[rf]{}$ (reads-from) relations that did not exist in any Observable Behavior of the original Candidate Execution. It is important to note however, that a proof in this sense would be generalized to any Candidate and is thus conservative. So, it might be the case that for specific programs, reordering can be valid, however, in a general sense may not be valid for others.

Assumption We make the following assumptions for every program we consider :

1. All events are tear-free
2. No synchronize events exist
3. No Read-Modify-Write events exist
4. All executions of the candidate before reordering have happens-before as a strict partial order

We first consider when consecutive events in the same agent can be reordered, followed by non-consecutive cases. The crux of the proof is to guarantee that reordering does not bring any new reads-from relations that did not result due to any execution of the original program.

The following definitions and lemmas are not particular to instruction reordering, so I think we can make it a point to put this in a section that introduces our work on optimizations.

5.1.1 Preliminaries

Before we go about proving when reordering is valid, we would like to have two additional definitions which would prove useful.

Definition 6. *Consecutive pair of events (*cons*)*

We define cons *as a function, which takes two events as input, and gives us a boolean indicating if they are consecutive pairs. Two events e and d are consecutive if they have an $\xrightarrow[ao]{}$ relation among them and are* next to each other*, which can be defined formally as*

$$(e \xrightarrow[ao]{} d \ \land \ \nexists k \ s.t. \ e \xrightarrow[ao]{} k \ \land \ k \xrightarrow[ao]{} d) \ \lor \ (d \xrightarrow[ao]{} e \ \land \ \nexists k \ s.t. \ d \xrightarrow[ao]{} k \ \land \ k \xrightarrow[ao]{} e)$$

Definition 7. *Direct happens-before relation (dir)*

We define dir *to take an ordered pair of events* (e, d) *such that* $e \xrightarrow[hb]{} d$ *and gives a boolean value to indicate whether this relation is direct, i.e those relations that are not derived through transitive property of* $\xrightarrow[hb]{}$.

We can infer certain things using this function based on some information on events e and d.

- *If* $e\!:uo$*, then* $dir(e, d) \;\Rightarrow\; cons(e, d)$
- *If* $d\!:uo$*, then* $dir(e, d) \;\Rightarrow\; cons(e, d)$
- *If* $e\!:sc \;\wedge\; e{\in}R$*, then* $dir(e, d) \;\Rightarrow\; cons(e, d)$
- *If* $e\!:sc \;\wedge\; e{\in}W$*, then* $dir(e, d) \;\Rightarrow\; cons(e, d) \;\vee\; e \xrightarrow[sw]{} d$
- *If* $d\!:sc \;\wedge\; d{\in}W$*, then* $dir(e, d) \;\Rightarrow\; cons(e, d)$
- *If* $d\!:sc \;\wedge\; e{\in}R$*, then* $dir(e, d) \;\Rightarrow\; cons(e, d) \;\vee\; e \xrightarrow[sw]{} d$

5.1.2 Lemmas to assist our proof

In order to assist our proof, we define two *lemmas* based on the ordering relations.

Lemma 1. *Consider three events e,d and k.*

If

$$cons(e, d) \;\wedge\; e \xrightarrow[ao]{} d \;\wedge\; ((d\!:uo) \;\vee\; (d\!:sc \;\wedge\; d{\in}W))$$

then,

$$k \xrightarrow[hb]{} d \Longrightarrow k \xrightarrow[hb]{} e$$

When we have two consecutive events e and d which are one after the other (i.e. $e \xrightarrow[ao]{} d$*), we can use transitive property of* $\xrightarrow[hb]{}$ *to infer that any event k that happens before e, also happens before d. However, is it possible to derive that the event k happens before e using the evidence that k happens before d ? This lemma states the condition when this is true.*

Proof. We will divide the proof for this into two cases, based on what event d is. For both cases, we have the following to be true :

$$cons(e, d) \;\wedge\; e \xrightarrow[ao]{} d \tag{0}$$

In the first case,

$$d\colon uo \tag{1}$$

Then for any event k

$$dir(k,d) \Rightarrow cons(k,d) \qquad from \quad (1) \tag{2}$$

An event that satisfies the above with d is e.

$$k = e \qquad from \quad (0,2) \tag{3}$$

Because $\xrightarrow[ao]{}$ is a total order, e will be the only event. This would mean that for any other $k \neq e$,

$$k \xrightarrow[hb]{} d \Rightarrow k \xrightarrow[hb]{} d \qquad from \quad (0,1,2,3)$$

The following figure should explain this intuition:

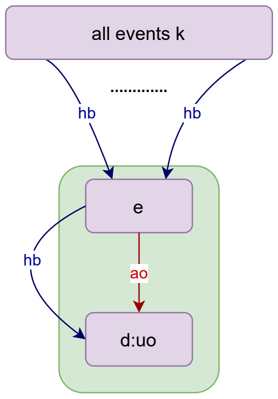


Figure 5.1: For the first case

In the second case,

$$d\colon sc \wedge d \in W \tag{4}$$

Then for any event k

$$dir(k,d) \Rightarrow cons(k,d) \qquad from \quad (4) \tag{5}$$

We once again have event e satisfying the above

$$k = e \qquad from \quad (0,5) \tag{6}$$

Though there could be direct *happens-before* relation with some event k from another *agent*, these are only relations satisfying $dir(d,k)$. Thus, we can once again infer that for any $k \neq e$

$$k \underset{hb}{\overrightarrow{\quad}} d \Rightarrow k \underset{hb}{\overrightarrow{\quad}} d \qquad from \quad (0,4,5,6)$$

The following figure explains this intuition:

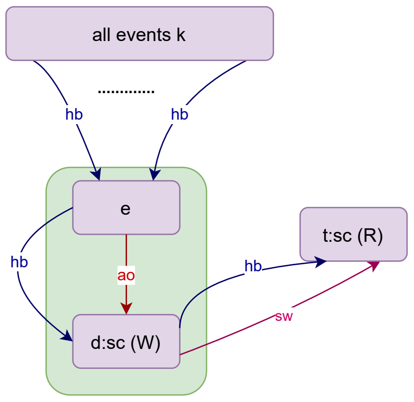


Figure 5.2: For the second case

□

Lemma 2. *Consider three events e, d and k*

If

$$cons(e,d) \ \wedge \ e \underset{ao}{\overrightarrow{\quad}} d \ \wedge \ ((e\!:\!uo) \ \vee \ (e\!:\!sc \ \wedge \ e\!\in\!R))$$

then,

$$e \underset{hb}{\overrightarrow{\quad}} k \Longrightarrow d \underset{hb}{\overrightarrow{\quad}} k$$

When we have two consecutive events e and d which are one after the other (i.e. $e \underset{ao}{\overrightarrow{\quad}} d$), we can use transitive property of $\underset{hb}{\overrightarrow{\quad}}$ to infer that any event k that happens after d, also happens after e. However, is it possible to derive that the event k happens after d using the evidence that k happens after e ? This lemma states the condition when this is true.

Proof. Just like the proof for the previous lemma, we will divide the proof for this into two cases, based on what event e is. Again, for both cases, we have the following to be true:

$$cons(e,d) \ \wedge e \xrightarrow[ao]{} d \tag{0}$$

In the first case,

$$e : uo \tag{1}$$

Then for any event k

$$dir(e,k) \Rightarrow cons(e,k) \qquad from \quad (1) \tag{2}$$

The event that satisfies the above with e is d

$$k = d \qquad from \quad (0,2) \tag{3}$$

Because $\xrightarrow[ao]{}$ is a total order, d would be the only such event. This would mean that for any other event $k \neq d$

$$e \xrightarrow[hb]{} k \Rightarrow d \xrightarrow[hb]{} k \qquad from \quad (0,1,2,3)$$

The following figure should explain this intuition:

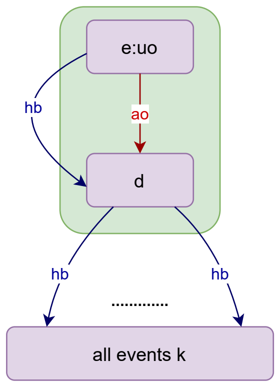


Figure 5.3: Caption

In the second case,

$$e : sc \wedge e \in R \tag{4}$$

Then for any event k

$$dir(e,k) \Rightarrow cons(e,k) \qquad from \quad (4) \tag{5}$$

We once again have event d satisfying the above

$$k = d \qquad from \quad (0, 5) \tag{6}$$

Though there could be direct *happens-before* relation with some event k from another *agent*, these are only relations satisfying $dir(k, e)$. Thus, we can once again infer that for any $k \neq d$

$$e \xrightarrow[hb]{} k \Rightarrow d \xrightarrow[hb]{} k \qquad from \quad (0, 4, 5, 6)$$

The following figure explains this intuition:

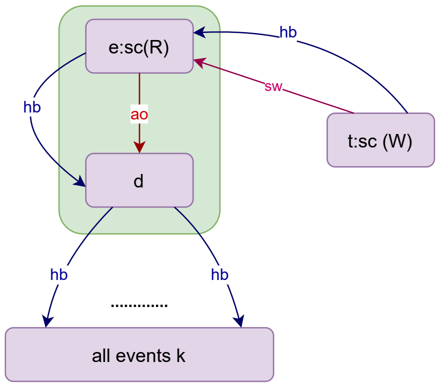


Figure 5.4: Caption

$\square$

5.2 Valid reordering

We view reordering as manipulating the agent-order relation among two events. In that sense, reordering two consecutive events e and d such that $e \xrightarrow[ao]{} d$ becomes:

$$e \xrightarrow[ao]{} d \longmapsto d \xrightarrow[ao]{} e$$

What implications this change has on the other ordering relations depends on the type of events e and d are and would require an analysis on each Candidate Execution. The intuition is that the axioms of the memory model rely on certain ordering relations to restrict observable behaviors in a program. Hence, preserving these ordering relations would help us in turn not introduce new Observable Behaviors. In particular we note that preserving $\xrightarrow[hb]{}$ relations (other than the one we eliminate intentionally i.e $e \xrightarrow[hb]{} d$) would suffice for our purpose. Since $\xrightarrow[mo]{}$ respects $\xrightarrow[hb]{}$, we in turn even preserve the memory order which is essential.

In the end, we want to ensure that the set of possible observable behaviors of a program, remain unchanged after reordering. If that is not feasible, then we would want the set of observable behaviors after reordering at the very least to be a subset. This would ensure that the program does not have some new behaviours that weren't supposed to happen prior to reordering.

We begin by first defining a reorderable pair of events. We then formulate a theorem (with a proof) on the set of observable behaviors of a Candidate before and after reordering a pair of consecutive events which are reorderable. We consider reordering valid if the set of observable behaviours after reordering are a subset of the original.

Definition 8. *Reorderable Pair (Reord) We define a boolean function* Reord *that takes two ordered pair of events e and d such that $e \xrightarrow[ao]{} d$ and gives a boolean value indicating if they are a reorderable pair.*

$$
\begin{aligned}
Reord(e,d) = &\\
&(((e\!:\!uo \ \wedge \ d\!:\!uo) \ \wedge \ ((e\!\in\!R \ \wedge \ d\!\in\!R) \ \vee \ (\Re(e) \cap_{\Re} \Re(d) = \phi)))\\
&\vee\\
&((e\!:\!sc \ \wedge \ d\!:\!uo) \ \wedge \ ((e\!\in\!W \ \wedge \ (\Re(e) \cap_{\Re} \Re(d) = \phi))))\\
&\vee\\
&((e\!:\!uo \ \wedge \ d\!:\!sc) \ \wedge \ ((d\!\in\!R \ \wedge \ (\Re(e) \cap_{\Re} \Re(d) = \phi)))))
\end{aligned}
$$

Use the latter for the purpose at the end of the proof for reordering, to emphasize how we approached each case

Theorem 2.1. *Consider a candidate C of a program and its possible Candidate Executions where $\xrightarrow[hb]{}$ is strictly partial order. Consider two events e and d such that cons(e,d) is true in C and $e \xrightarrow[ao]{} d$. Consider another candidate C' resulting after reordering e and d. Then if* Reord(e,d) *is true in C, the set observable behaviors possible due to Candidate Executions of C' is a subset of that of C.*

Proof. We look at this in terms of performing an instruction reordering on a candidate execution of C. We would want the resulting candidate execution to preserve all the other $\xrightarrow[hb]{}$ relations (except $e \xrightarrow[hb]{} d$) and that any new $\xrightarrow[hb]{}$ relations strictly reduce possible observable behaviors.

The proof is structured as follows. We first show that existing *happens-before* relations in any candidate execution of C except $e \xrightarrow[hb]{} d$ remain intact after reordering. We then identify the cases where new *happens-before* relations could be established. We identify from these cases whether *happens-before* cycles could be introduced. We then show for the remaining cases that new relations do not introduce any new observable behaviors.

The above steps can be summarized as addressing four main questions for any $CandidateExecution$ of C'

1. Apart from $e \xrightarrow[hb]{} d$, do other *happens-before* relations remain intact?
2. Apart from $d \xrightarrow[hb]{} e$, are any new *happens-before* relations established?
3. Are any *happens-before* cycles introduced?
4. Do the new relations bring new *observable behaviors?*

1. Preserving *happens-before* relations If $\xrightarrow[hb]{}$ relations among events are lost after reordering, we may introduce new observable behaviors. The relations that are subject to change can be divided into four parts using events e and d

a) $k \xrightarrow[hb]{} e$

b) $e \xrightarrow[hb]{} k$

c) $d \xrightarrow[hb]{} k$

d) $k \xrightarrow[hb]{} d$

Firstly, note that the relations of the form $e \xrightarrow[hb]{} k$ come through either a $\xrightarrow[sw]{}$ relation with e or relations through event d, i.e. of the form $d \xrightarrow[hb]{} k$. The ones that come due to the latter, may not be preserved after reordering, if we strictly are only able to derive them with relations through d. Note also that, a similar argument exists for relations of the form $k \xrightarrow[hb]{} d$ wherein relations derived through e ($k \xrightarrow[hb]{} e$) may be lost after reordering.

Hence, the relations that could be subject to change can be addressed by considering two disjoint sets of events in any *Candidate Execution* of C as below.

$$K_e = \{k \mid k \xrightarrow[hb]{} e\}.$$
$$K_d = \{k \mid d \xrightarrow[hb]{} k\}.$$

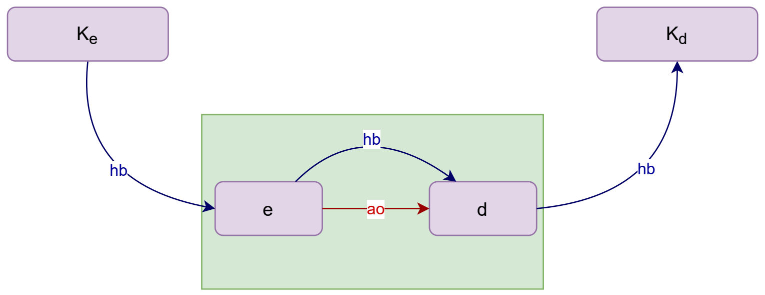


Figure 5.5: For any Candidate Execution of C, the set K_e and K_d

Consider two events $p1 \in K_e$ and $p2 \in K_d$ (When e is the first event or d is the last event, assume dummyevents that can act as $p1$ or $p2$.) belonging to the same agent as that of e and d such that in C:

$$dir(p1, e) \ \wedge \ dir(d, p2).$$

Note that in terms of direct happens-before relations, on reordering, any $CandidateExecution$ of C will have the followingchanges

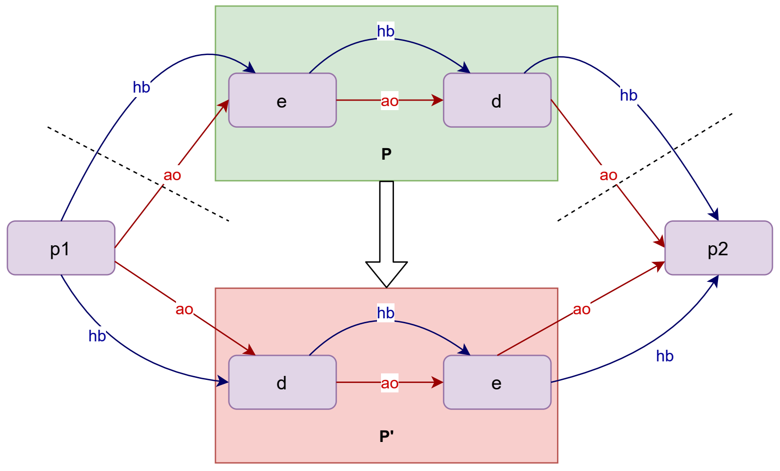


Figure 5.6: The direct relation changes that can be observed while reordering events e and d

The figure above is to show that, for any $CandidateExecution$ of C, the following is true

$$cons(p1, e) \ \wedge dir(p1, e) \ \wedge dir(e, d) \ \wedge cons(d, p2) \ \wedge \ dir(d, p2).$$

and for that of C',

$$cons(p1, d) \ \wedge \ dir(p1, d) \ \wedge \ dir(d, e) \ \wedge cons(e, p2) \ \wedge dir(e, p2).$$

We need the following key relations to be preserved in Candidate executions of C'

a) $p1 \xrightarrow[hb]{} e$

b) $e \xrightarrow[hb]{} k$

c) $d \xrightarrow[hb]{} p2$

d) $k \xrightarrow[hb]{} d$

After reordering, we do have these relations preserved due to transitivity

$$p1 \xrightarrow[hb]{} d \ \wedge \ d \xrightarrow[hb]{} e \ \Rightarrow \ p1 \xrightarrow[hb]{} e.$$
$$e \xrightarrow[hb]{} p2 \ \wedge \ d \xrightarrow[hb]{} e \ \Rightarrow \ d \xrightarrow[hb]{} p2.$$
$$p1 \xrightarrow[hb]{} d \ \wedge \ d \xrightarrow[hb]{} e \ \wedge \ e \xrightarrow[hb]{} p2 \ \Rightarrow \ p1 \xrightarrow[hb]{} p2.$$

The other two forms of relations may not be preserved due to $d \xrightarrow[sw]{} k$ or $k \xrightarrow[sw]{} d$. If we can "pivot" the set K_e to $p1$ and K_d to $p2$, it would ensure that our other two intended relations also remain preserved after reordering by transitivity. To state formally, we have a valid pair of pivots $< p1, p2 >$ when the following two conditions hold

$$\forall \ k \in K_e - \{p1\}, \ k \xrightarrow[hb]{} p1.$$
$$\forall \ k \in K_d - \{p2\}, \ p2 \xrightarrow[hb]{} k.$$

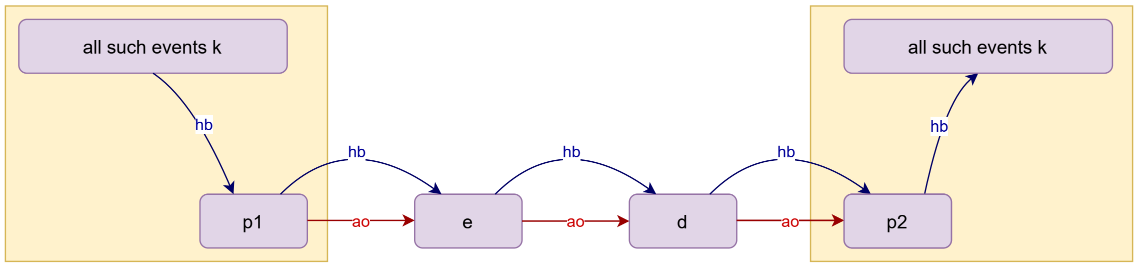
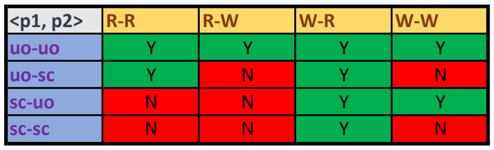


Figure 5.7: For any Candidate execution, the intuition behind valid pivots $< p1, p2 >$

By lemma 1 and lemma 2 respectively, we have for C, the following condition where $< p1, p2 >$ is a valid pivot pair

$$e\!:\!uo \vee (e\!:\!sc \wedge e \!\in\! W).$$
$$d\!:\!uo \vee (d\!:\!sc \wedge d \!\in\! R).$$

The following table summarizes the cases where we have a valid pair of pivots $< p1, p2 >$



<p1, p2>	R-R	R-W	W-R	W-W
uo-uo	Y	Y	Y	Y
uo-sc	Y	N	Y	N
sc-uo	N	N	Y	Y
sc-sc	N	N	Y	N

Figure 5.8: Table summarizing whether we have valid pair of pivots based on e and d

We show a simple example where we do not have a valid pair of pivots, particularly because $p1$ is not a valid pivot. Note thatin this example, $K_e = K_{e1} + K_{e2} + p1 + p_x$

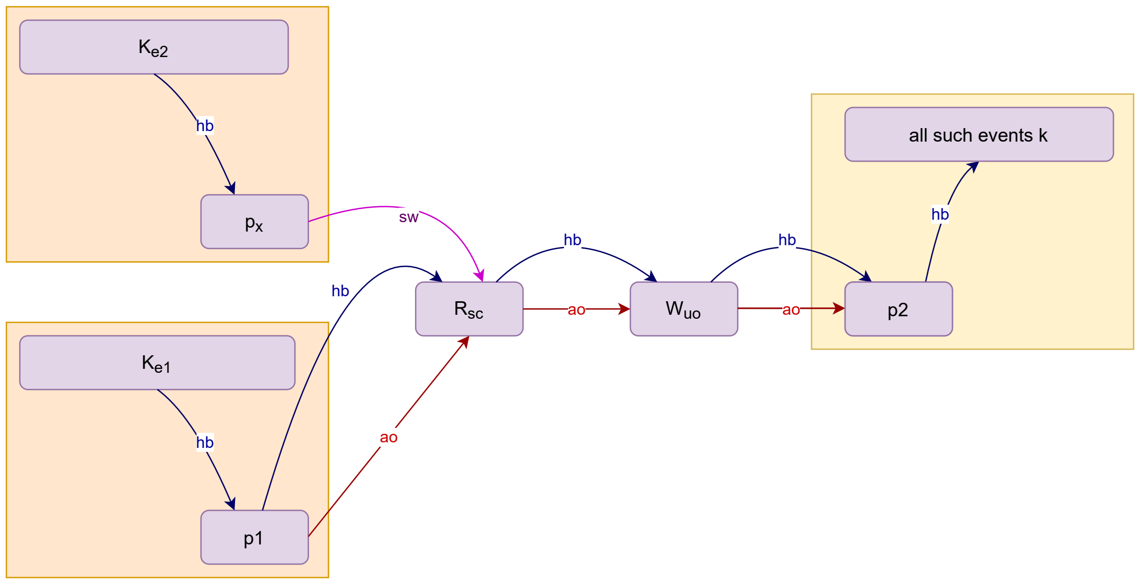


Figure 5.9: A Candidate Execution where p1 is not a valid pivot

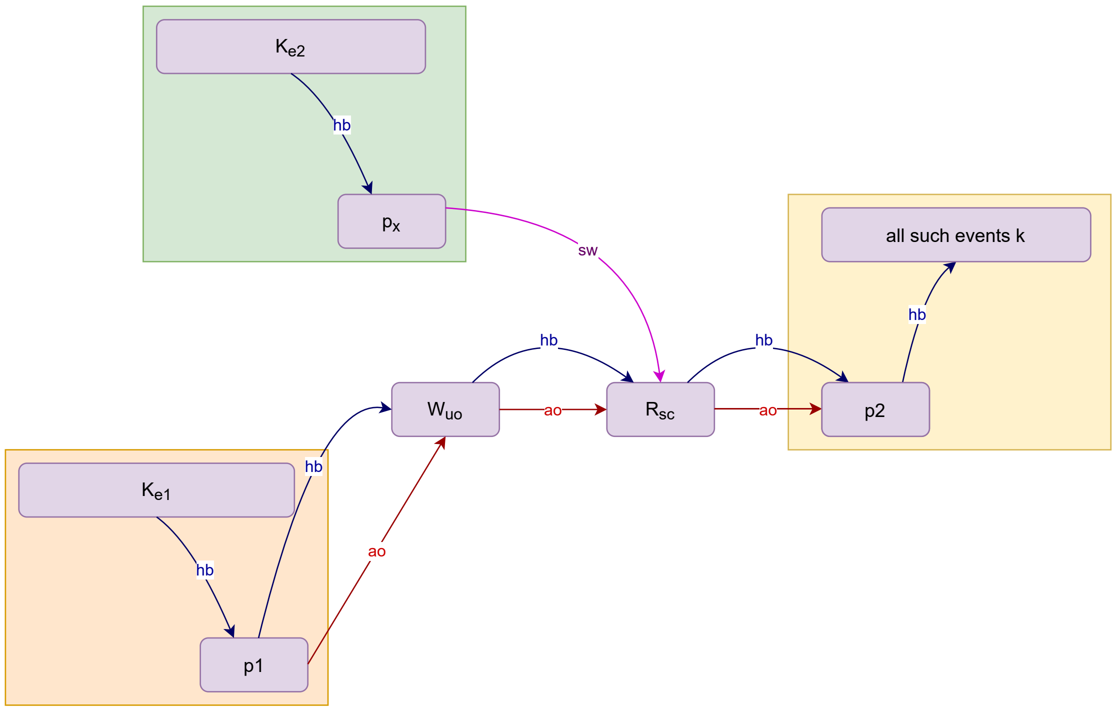


Figure 5.10: The resultant Candidate Execution after reordering, exposing the relations with p_x, K_{e2} and d that arelost

Strictly speaking, it is not that the happens-before relations are preserved, but that the properties between different happens before relations hold, which implies that for any possible Candidate Execution after reordering, the set of happens-before relatiosn apart from that between e and d remain the same. I am emphasizing this point because we view reordering as just changing agent order between two events, which just needs information from Candiadate but not all its possible Executions.

2. Additional *happens-before* relations Although we have identified the cases when *happens-before* relations are preserved, we also get some additional relations in some of them.

As an example, for the case when d is a sequentially consistent read, by lemma 1, in any execution of C

$$k \xrightarrow[hb]{} d \not\Rightarrow k \xrightarrow[hb]{} e$$

But in *Executions* of candidate C', by transitivity, we have

$$k \xrightarrow[hb]{} d \Rightarrow k \xrightarrow[hb]{} e$$

This is because, there are sets of relations that come through *synchronize-with* relations that d has. Thus, although we are able to preserve relations that existed in any $CandidateExecution$ of C, we also in the process, introduce new ones in $CandidateExecutions$ of C'. The figure below shows pictorially an example of a Candidate Execution of C for the case above

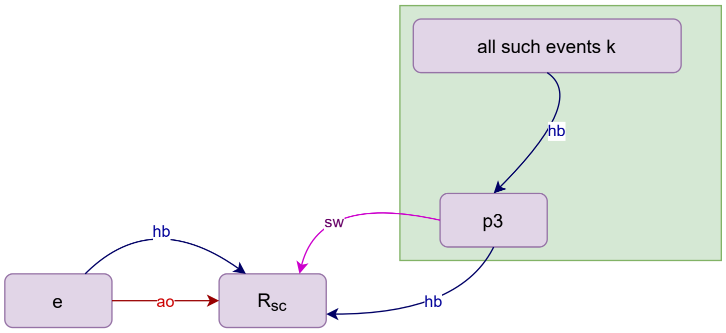


Figure 5.11: A Candidate Execution where d is a sequentially consistent read

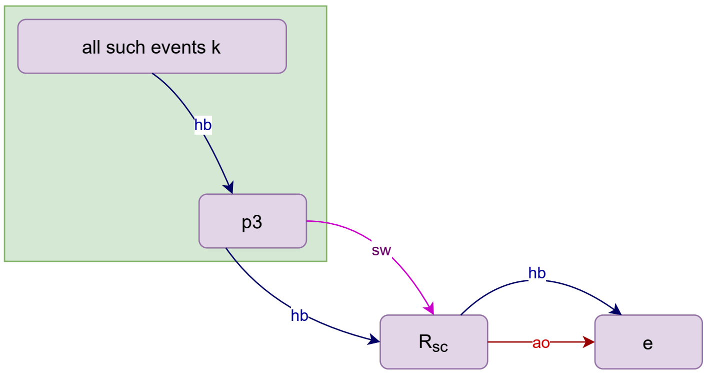
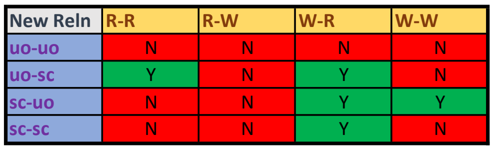


Figure 5.12: The Candidate Execution after reordering, exposing the new relations established with e, $p3$ and set k

Explain the above figures or perhaps highlight the new relations that are established.

To summarize, the table below shows the cases where new relations could be introduced.



New Reln	R-R	R-W	W-R	W-W
uo-uo	N	N	N	N
uo-sc	Y	N	Y	N
sc-uo	N	N	Y	Y
sc-sc	N	N	Y	N

Figure 5.13: Table summarizing when new *happens-before* relations could be introduced based on having valid pair of pivots

For these cases, we must know whether these new relations introduce new observable behaviors.

3. Presence of cycles? Before we go into analyzing whether new relations introduce observable behaviours, we first ensure there are no $\xrightarrow[hb]{}$ cycles introduced in the process. Consider the example below

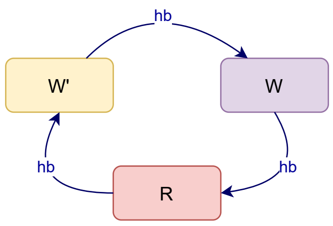


Figure 5.14: Caption

Notice that here, the axiom of coherent reads restricts R to read from W'.

$$R \xrightarrow[hb]{} W' \Rightarrow \neg R \xrightarrow[rf]{} W'$$

But by transitive property, it is also the case that $W' \xrightarrow[hb]{} R$.

$$W' \xrightarrow[hb]{} W \;\wedge\; W \xrightarrow[hb]{} R \;\Rightarrow\; W' \xrightarrow[hb]{} R$$

As per this, the axiom of coherent reads shouldn't restrict $R \xrightarrow[rf]{} W'$. To avoid such cases, we will need to ensure that no Candidate Execution of C' after e and d are reordered have $\xrightarrow[hb]{}$ cycles.

Note that if a cycle exists after reordering, then

1. The relations preserved do not themselves create a cycle (ref to the theorem)

2. Additional new relations may introduce cycles

The first part is straightforward as we assume we can only do reordering on Candidate Exectuions of C not having happens-before cycles.

For the second part, we first address the cases where $d \xrightarrow[hb]{} e$ may be part of the cycle. The other event k, may be either from the set K_e, K_d or a new relation that is formed.

K_e and K_d only apart from the new relation because these are the only valid cases where happens-before relations are preserved after reordering. So we need not consider cases such thta $e \xrightarrow[hb]{} k$ or $k \xrightarrow[hb]{} d$ as the old relationsas they are covered by K_e and K_d.

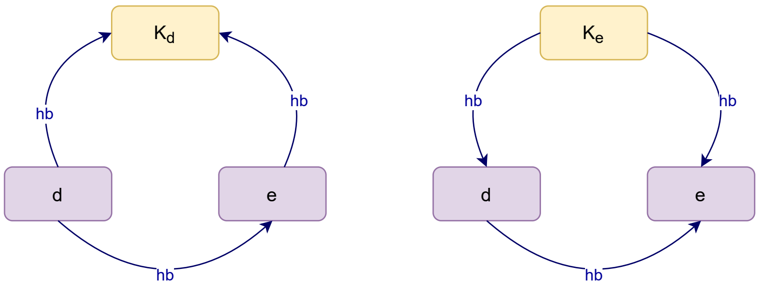


Figure 5.15: If k belongs to one of the sets K_e or K_d

The above figure shows that k cannot belong to either of the sets, as their relations with e and d will not result in a cycle.

For cases where $k \xrightarrow[hb]{} e$ is the set of new relations, note that by lemma 1

$$k \xrightarrow[hb]{} e \Rightarrow k \xrightarrow[hb]{} d$$

For cases where $d \xrightarrow[hb]{} k$ is the set of new relations, by lemma 2

$$d \xrightarrow[hb]{} k \Rightarrow e \xrightarrow[hb]{} k$$

So for both these cases also, a cycle with $d \xrightarrow[hb]{} e$ cannot exist. The following figure shows pictorially this fact.

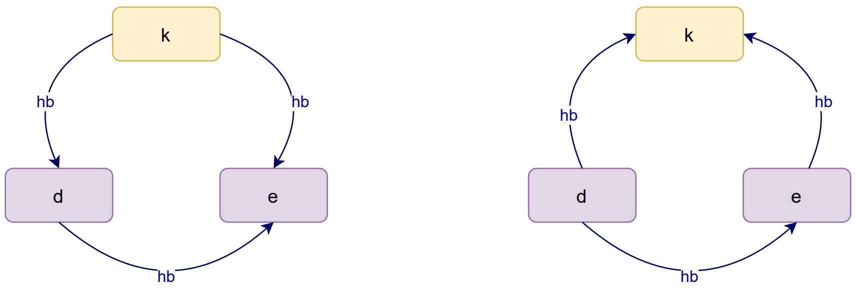


Figure 5.16: If $k \xrightarrow[hb]{} e$ or $d \xrightarrow[hb]{} k$ are new sets of relations

For the one case where we have two new sets of relations formed, i.e $d \xrightarrow[hb]{} k$ and $k \xrightarrow[hb]{} e$, we could have a case where k is a common event for both sets. But, by lemma 1, we also have $k \xrightarrow[hb]{} d$ and by lemma 2, $e \xrightarrow[hb]{} k$. Thus, we have a cycle. The following figure shows this pictorially

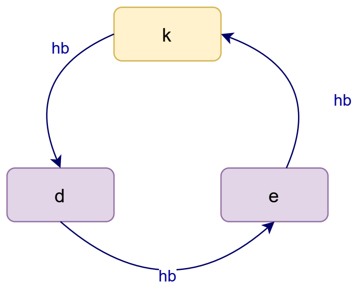


Figure 5.17: A cycle exists in the case where we have two new sets of relations ($k \xrightarrow[hb]{} e$ and $d \xrightarrow[hb]{} k$)

It is not actually due to lemmas, but just that the new relations were derived through e or d, as these relations existed before reordering.

Maybe have a better figure, meaning a set of relations where each figure shows clearly which relaiton is implied due to whcih lemma

Now for the case when $d \xrightarrow[hb]{} e$ may not be part of the cycle, we have only two other new relations, $k \xrightarrow[hb]{} e$ or $d \xrightarrow[hb]{} k$.

Considering the first scenario where the new set of relations are of the form $k \xrightarrow[hb]{} e$. Suppose a cycle exists with another event k'. Then

$$k \xrightarrow[hb]{} e \ \wedge \ e \xrightarrow[hb]{} k' \ \wedge \ k' \xrightarrow[hb]{} k$$

Note that the latter two relations are not new, since the only new set of relations are of the first form. Now, by lemma 1 and by transitivity respectively

$$k \xrightarrow[hb]{} e \Rightarrow k \xrightarrow[hb]{} d$$
$$e \xrightarrow[hb]{} k' \Rightarrow d \xrightarrow[hb]{} k'$$

So, the following is also a cycle

$$k \xrightarrow[hb]{} d \ \wedge \ d \xrightarrow[hb]{} k' \ \wedge \ k' \xrightarrow[hb]{} k$$

But these relations already exist in the original Candidate Execution, which implies a cycle existed before reordering. This contradicts our assumption that we only reorder when the Candidate Executions of C have no cycles. Thus, by contradiction such a cycle cannot exist.

In similar lines for the cases where the set of new relations are of the form $d \xrightarrow[hb]{} k$, we can show by contradiction that a cycle cannot exist.

4. Do new relations introduce new observable behaviors? In any candidate execution, reordering events e and d eliminates the relation $e \xrightarrow[hb]{} d$ and introduces the new relation $d \xrightarrow[hb]{} e$. New behaviours created by the latter directly, if any, are of course intentional (and should normally be avoided by ensuring e and d are independent), but we need to ensure that this does not also result in new behaviours indirectly.

On observing the role on the axioms on this relation, notice that if both e and d are read events then the range does not matter. For all other cases, if events e and d have overlapping ranges, one could introduce a new observable behavior afterreordering them (a simple use of Coherent Reads / Sequentially Consistent Atomics).

We will later show counter examples for each of the cases that we discard as invalid to reorder.

Any other new relations that are introduced can be divided into 4 cases, in terms of our events e and d and the newrelation with some event k:

a) $e\!:\!uo \;\wedge\; e\!\in\! R \;\wedge\; k \xrightarrow[hb]{} e.$

b) $e\!:\!uo \;\wedge\; e\!\in\! W \;\wedge\; k \xrightarrow[hb]{} e.$

c) $d\!:\!uo \;\wedge\; d\!\in\! R \;\wedge\; d \xrightarrow[hb]{} k.$

d) $d\!:\!uo \;\wedge\; d\!\in\! W \;\wedge\; d \xrightarrow[hb]{} k.$

Change the figure above to represent only the first four cases In each of the above cases, note firstly that we need to only consider cases where their ranges are overlapping/equal.

Figure below shows a breakdown of sub-cases for the first case (a), varying based on the nature of event k.

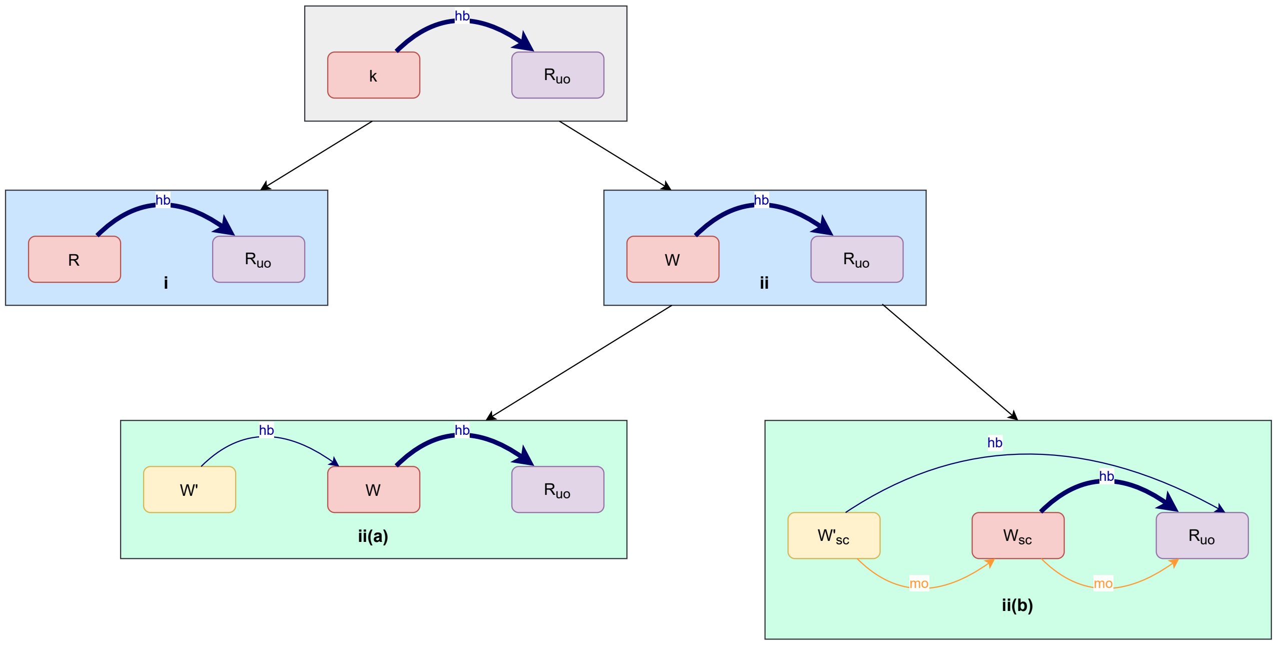


Figure 5.18: The role of the axioms on introducing a new relation between an unordered Read and some event k

1. For (i), when k is a read, none of the rules have any implications on observable behaviors.

2. For (ii), when k is a write, the rule of coherent reads (ii(a)) or sequentially consistent atomics (ii(b)) could restrict the read (e) from reading overlapping ranges of W' with W.

Figure below shows a breakdown of sub-cases for the first case (b), varying based on the nature of event k.

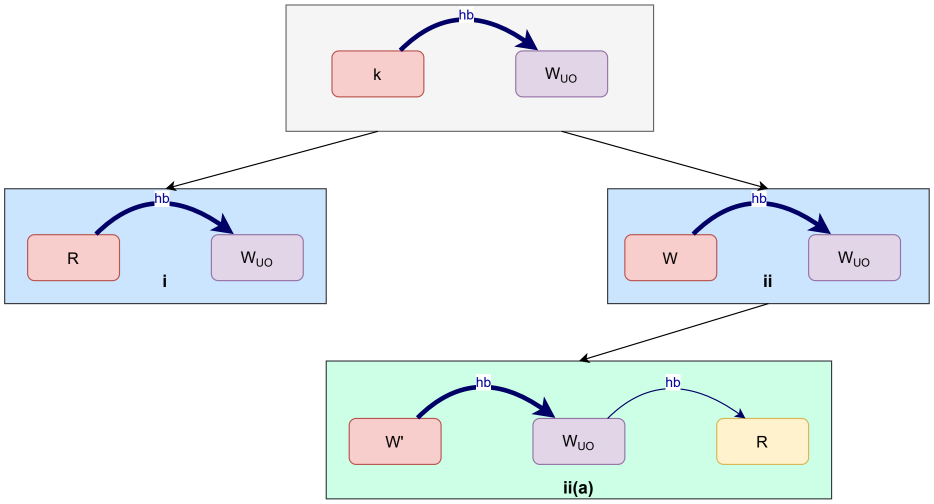


Figure 5.19: (i) and (ii(b)) satisfy the axiom of Coherent Reads

For case (b) we can observe the following from the above figure

1. For (i), when k is a read, the pattern of coherent reads restricts the k from reading from the write e.
2. For (ii), when k is a write, the pattern of coherent reads, restricts some read from reading parts of k due to the write e.

Figure below shows a breakdown of sub-cases for the first case (c), varying based on the nature of event k.

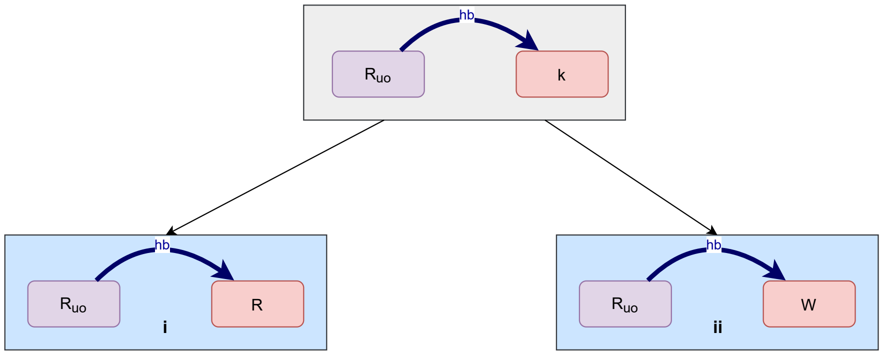


Figure 5.20: (ii) satisfies the axiom of Coherent Reads

For case (c), we can observe the following from the above figure

1. Case (i) does not correspond to any pattern restricted on the model, thus having no impact on the observable behaviors.

2. Casee (ii) is a pattern of coherent reads, which restricts the read d from reading values from the write.

Figure below shows a breakdown of sub-cases for the first case (d), varying based on the nature of event k.

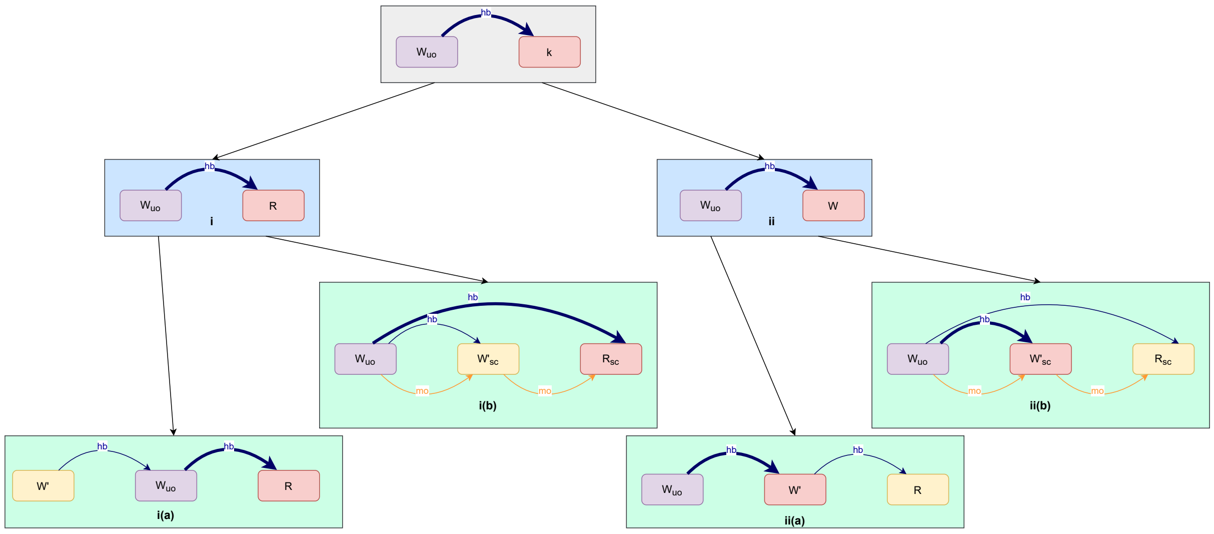


Figure 5.21: (i(a)), (ii(a)) satisfy the axiom of Coherent Reads, whereas (i(b)), (ii(b)) satisfy the axiom of SequentiallyConsistent Atomics

For case (d) we can observe the following

1. For case (i), the pattern of coherent reads (i(a)) or the pattern of sequentially consistent atomics (i(b)) could restrict a read from reading values of write d,

2. For case (ii), the pattern of coherent reads (ii(a)) or the pattern of sequentially consistent atomics (ii(b)) could restrict a read from reading values of write d,

The above case analysis shows us that the new relation could ’trigger’ the consistency rules, only to restrict possible reads-from relations, thus restricting possible observable behaviors. For all cases, instances which can ’trigger’ some cases of the consistency rules only restrict possibly some $\overrightarrow{rf}$ relations. Thus, we can conclude that no new observable behavior is introduced due to the new set of $\overrightarrow{hb}$ relations.

The main reason for this is that we framed he axioms in a form that restricts *reads-from* relations. Soin any case where adding an additional *happens-before* relation ”triggers” an axiom, we are bound to have somebehaviors restricted. It is this fact that is elicited explicitly by going case wise on all relations that are introduced.

The table below summarizes the valid cases where, we have a pair of valid pivots, where new relations do not introduce new observable behaviors and do not have cycles.

Final	R-R	R-W	W-R	W-W
uo-uo	Y	Y	Y	Y
uo-sc	Y	N	Y	N
sc-uo	N	N	Y	Y
sc-sc	N	N	N	N

Figure 5.22: The final table summarizing the valid cases where observable behaviors will only be a subset after reordering.

Keep in mind that the comparision of ranges is done while addressing question 3 in the proof, so the table above, implicitly also takes into account only the valid cases where ranges are also correct

The table above, precisely is the definition of a reorderable pair. If we write the above table in the form of an expression we have an expanded format of our Reorderable pair function.

$$
\begin{aligned}
&Reord(e,d) = \\
&(((e\!:uo \wedge d\!:uo) \wedge \\
&\qquad ((e\!\in\! R \wedge d\!\in\! R)\vee \\
&\qquad\ (e\!\in\! W \wedge d\!\in\! R \wedge (\Re(e)\ \ \&\ \ \Re(d) = \phi))\vee \\
&\qquad\ (e\!\in\! R \wedge d\!\in\! W \wedge (\Re(e)\ \ \&\ \ \Re(d) = \phi))\vee \\
&\qquad\ (e\!\in\! W \wedge d\!\in\! W \wedge (\Re(e)\ \ \&\ \ \Re(d) = \phi)))) \\
&\qquad \vee \\
&((e\!:sc \wedge d\!:uo) \wedge \\
&\qquad ((e\!\in\! W \wedge d\!\in\! R \wedge (\Re(e)\ \ \&\ \ \Re(d) = \phi))\vee \\
&\qquad\ (e\!\in\! W \wedge d\!\in\! W \wedge (\Re(e)\ \ \&\ \ \Re(d) = \phi)))) \\
&\qquad \vee \\
&((e\!:uo \wedge d\!:sc) \wedge \\
&\qquad ((e\!\in\! R \wedge d\!\in\! R\wedge)\vee \\
&\qquad\ (e\!\in\! W \wedge d\!\in\! R \wedge (\Re(e)\ \ \&\ \ \Re(d) = \phi)))))
\end{aligned}
$$

□ □

Corollary 2.1.1. *Consider a Candidate C of a program and its Candidate Executions which are valid. Consider two events e and d such that $\neg cons(e,d)$ is true in C and $e \underset{ao}{\rightarrow} d$. Consider another Candidate C' resulting after reordering e and d in C. Then, the set of Observable behaviors possible in C' is a subset of C only if $Reord(e,d)$ and the following holds true.*

$$\forall\ k\ s.t.\ e \underset{ao}{\rightarrow} k\ \wedge\ k \underset{ao}{\rightarrow} d\ .\ Reord(e,k)\ \wedge\ Reord(k,d)$$

Proof. We prove this by induction of number of events k between e and d. Let n denote the number of events.

Base Case: $n = 1$. This means we have one event k such that

$$e \underset{ao}{\rightarrow} k \underset{ao}{\rightarrow} d$$

What we want after reordering is

$$d \underset{ao}{\rightarrow} k \underset{ao}{\rightarrow} e$$

Without loss of generality, we can choose to first reorder k and d. For this we have $cons(k, d)$ to be true. To reorder, what we only need is $Reord(k, d)$ to hold. Thus, after reordering, we have

$$e \xrightarrow[ao]{} d \xrightarrow[ao]{} k$$

Similarly, now we need $Reord(e, d)$ to hold to reorder them above, after doing so, we will get

$$d \xrightarrow[ao]{} e \xrightarrow[ao]{} k$$

Now lastly, we need to reorder e and k for which we need $Reord(e, k)$ to hold, thus giving us our final result

$$d \xrightarrow[ao]{} k \xrightarrow[ao]{} e$$

By transitive property of subsets, we can conclude that the Observable Behavior of the final program after reordering is a subset of the original.

2. Inductive Case $n > 1$ Assume the above corollary holds for $n = t$.

We need to show that for $n = t + 1$, the corollary still holds, for this note firstly that, we have the following ordering relations.

$$e \xrightarrow[ao]{} k_1 \xrightarrow[ao]{} k_2 \xrightarrow[ao]{} k_3 \xrightarrow[ao]{} ... \xrightarrow[ao]{} k_t \xrightarrow[ao]{} k_{t+1} \xrightarrow[ao]{} d$$

Without loss of generality, we can first reorder k_{t+1} and d. To do this, we need $Reord(k_{t+1}, d)$ to hold, thus giving us the resultant ordering.

$$e \xrightarrow[ao]{} k_1 \xrightarrow[ao]{} k_2 \xrightarrow[ao]{} k_3 \xrightarrow[ao]{} ... \xrightarrow[ao]{} k_t \xrightarrow[ao]{} d \xrightarrow[ao]{} k_{t+1}$$

Now we have t such events between e and d. With our assumption, we can reorder e and d, thus giving us

$$d \xrightarrow[ao]{} k_1 \xrightarrow[ao]{} k_2 \xrightarrow[ao]{} k_3 \xrightarrow[ao]{} ... \xrightarrow[ao]{} k_t \xrightarrow[ao]{} e \xrightarrow[ao]{} k_{t+1}$$

Finally, we need to reorder e and k_{t+1} to get our final result, for which we need $Reord(e, k_{t+1})$ to hold, thus giving us finally

$$d \xrightarrow[ao]{} k_1 \xrightarrow[ao]{} k_2 \xrightarrow[ao]{} k_3 \xrightarrow[ao]{} ... \xrightarrow[ao]{} k_t \xrightarrow[ao]{} k_{t+1} \xrightarrow[ao]{} e$$

Thus we can see that we need two more conditions to hold for us to ensure we can reorder e amd d with $n = t + 1$. by transitive property of subsets, we can conclude that the Observable Behavior of the final program after reordering is a subset of the original.

Hence, by induction the proof is complete.

Observe that wherever our argument states the requirement of *Reord* to hold between two events, it is also the case that those two events are consecutive and have an agent ordering exactly as our Theorem states.

□

To investigate the validity of reordering at the program level, we first, in terms of candidates, address code motion. The following two corollaries cover them:

Corollary 2.1.2. *Consider a Candidate C of a program and its Candidate Executions which are valid. Consider a set of events $k_{i\in[1,n]}$ such that $k_i \underset{ao}{\rightarrow} k_{i+1} \wedge cons(k_i, k_{i+1})$. Consider an event e such that*

$$cons(e, k_1) \;\wedge\; e \underset{ao}{\rightarrow} k_1$$

Consider another candidate C' with the only differnence from C being $cons(e, k_n) \wedge k_n \underset{ao}{\rightarrow} e$. Then the set of observable behaviors of C' is a subset of that of C if

$$\forall i \in [1, n] \;.\; Reord(e, k_i) \tag{5.1}$$

Proof. Apply theorem of reordering successively, and by transititvity of subset relations, the corollary holds. □

Corollary 2.1.3. *Consider a Candidate C of a program and its Candidate Executions which are valid. Consider a set of events $k_{i\in[1,n]}$ such that $k_i \underset{ao}{\rightarrow} k_{i+1} \wedge cons(k_i, k_{i+1})$. Consider an event d such that*

$$cons(d, k_n) \;\wedge\; k_n \underset{ao}{\rightarrow} d$$

Consider another candidate C' with the only differnence from C being $cons(d, k_1) \wedge d \underset{ao}{\rightarrow} k_1$. Then the set of observable behaviors of C' is a subset of that of C if

$$\forall i \in [1, n] \;.\; Reord(k_i, d) \tag{5.2}$$

Proof. Apply theorem of reordering successively, and by transititvity of subset relations, the corollary holds. □

5.2.1 Counter examples for the invalid cases

For each case where reordering is not safe to do, we also show counter examples of programs where new observable behaviors are introduced. This additionally potrays additional proof of the validity of our approach.

For all the examples we show here, we only show the ordering relations that are important to observe. Putting all the relations among different events in the example will result in confusion, hence we avoid doing so.

Reads to same memory where e is of type sc while d is of either uo/sc The following example involves two reads to the same memory and a write.

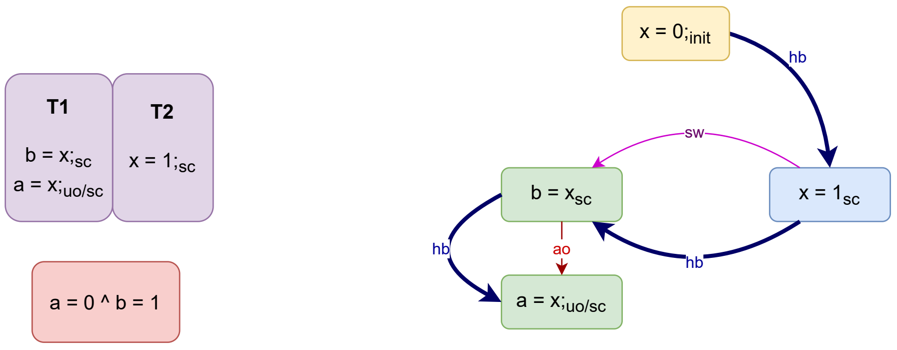


Figure 5.23: Case where a = 0 , b = 1 is invalid due to Coherent Reads

The figure on the left above shows an example of a candidate where the case of reads in the red box is not possible. The figure on the right shows the Candidate Execution of such a case. Observations:

- We can infer from the Candidate Execution that $\{x = 0_{init}\} \overrightarrow{hb} \{x = 1_{sc}\} \overrightarrow{hb} \{a = x_{uo/sc}\}$.
- By the second axiom of coherent reads, it is not possible for a to read the value of 0 as x due to the intervening write whiCh changes x to 1.
- This inference does not rely upon the access mode of the read a.

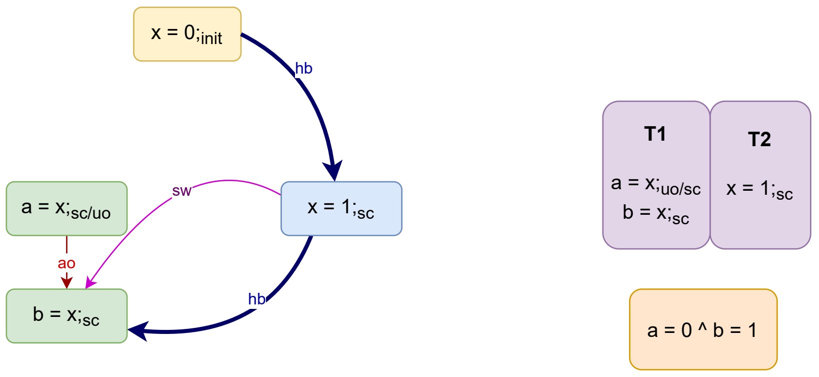


Figure 5.24: Case where the reads are reordered and a = 0 , b = 1 is valid

The figure on the right shows the program after reordering the two reads in $T1$, where the case of reads in the orange box is possible. The figure on the left shows the Candidate Execution of such a case.

Observations:

- From the Candidate Execution, we can infer $\neg\{x = 0_{init}\} \xrightarrow[hb]{} \{x = 1_{sc}\} \xrightarrow[hb]{} \{a = x_{uo/sc}\}$
- We can also infer that $\{x = 0_{init}\} \xrightarrow[hb]{} \{a = x_{uo/sc}\}$
- Since none of the axioms disallow the above pattern, a is allowed to read the value of x to be 0.
- Hence, the reordering of the two reads is invalid.

Reads to non-equal range of memory where e is of type sc while d is of either uo/sc

A Read e of type sc followed by a Write of either uo/sc The following is an example of a program with a sequentially consistent read followed by a write of any type.

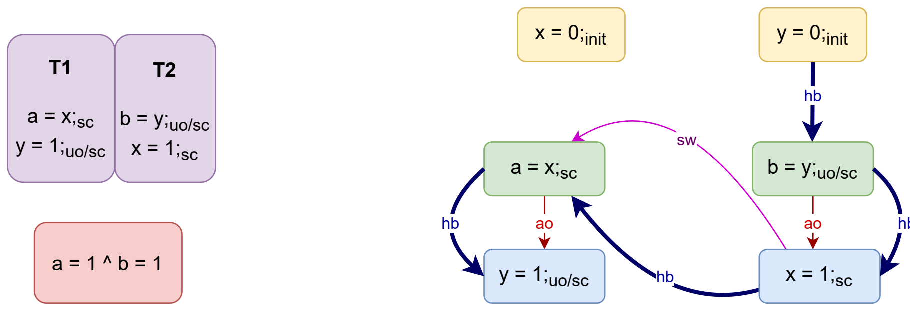
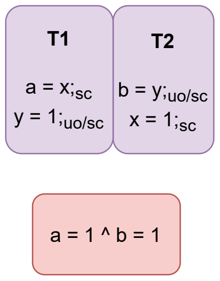


Figure 5.25: Case where a = 1 and b = 1 is invalid due to Coherent Reads.

The figure on the left above shows an example of a candidate where the case of reads in the red box is not possible. The figure on the right shows the Candidate Execution of such a case. Observations:

- From the Candidate Execution, we can infer $b = y_{uo/sc} \xrightarrow[hb]{} y = 1_{uo/sc}$
- By the first rule of coherent reads, b cannot read the value of 1 as y.

- This inference was due to $x = 1_{sc} \xrightarrow[hb]{} a = xsc$



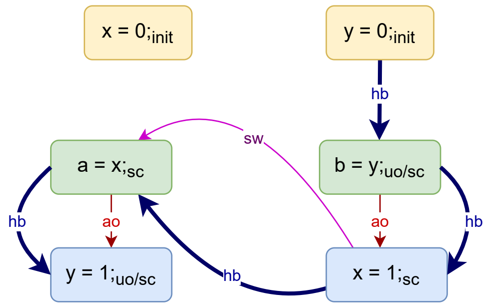
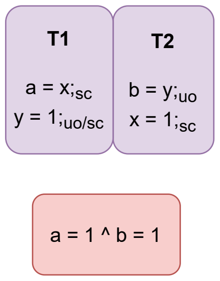


Figure 5.26: Case where events of T1 are reordered, resulting in a = 1 and b = 1 to be valid.

The figure on the right above shows the program after reordering the two events in $T1$ where case of reads in the orange box is possible. The figure on the left shows the Candidate Execution of such a case. Observations:

- From the Candidate Execution, we can infer $\neg b = y_{uo/sc} \xrightarrow[hb]{} y = 1_{uo/sc}$
- Since there is no $\xrightarrow[hb]{}$ relation among the above two events, b can read the value of y as 1.

A Read e of type uo followed by a write d of type sc For this we can use the same example for the previous part (tag figure of example), where we just reorder $T2$'s events.



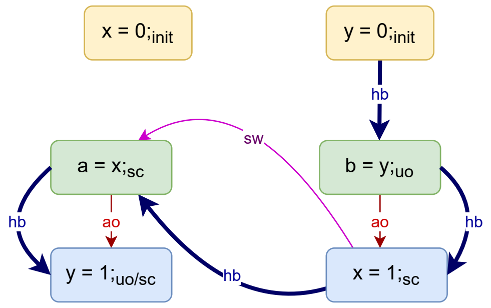


Figure 5.27: Case where a = 1 and b = 1 is invalid due to Coherent Reads.

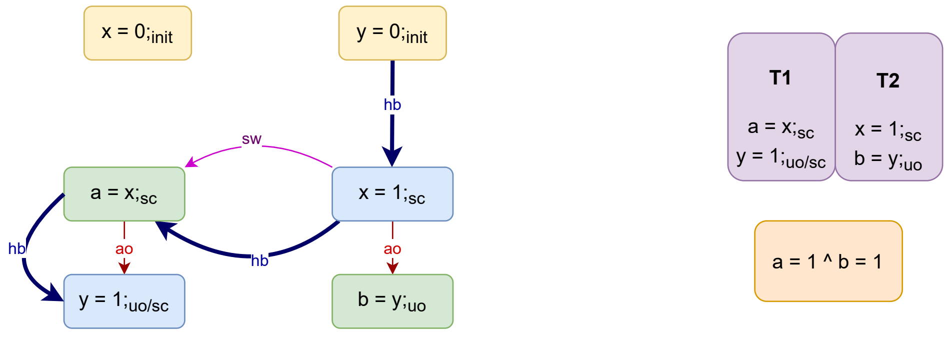


Figure 5.28: Case where events of T2 are reordered, resulting in a = 1 and b = 1 to be valid.

A Write e followed by a Read d both of type sc A counter example for this is different. It is not the Observable Behavior we are concerned with that is introduced, but that which is allowed but creates a $\overrightarrow{hb}$ cycle. The following example is as such:

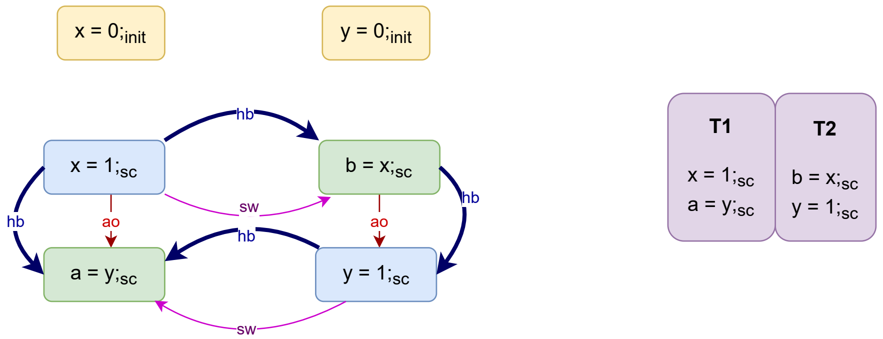


Figure 5.29: Case where a = 1 and b = 1 is valid and no happens-before cycles

After reordering the two events of $T1$ in the above example, the same observable behavior holds, but has a cycle introduced. One might think that simply discarding that execution would do. But this would mean discarding $\overrightarrow{hb}$ relations also, which would require more information to infer which relations are going to create such cycles and which are not. Since we place no assumptions on these relations, but that any happens-before relation other than the one we remove explicitly be reordering are all possible. Hence, the following reordered program outcome is something we do not risk to allow.

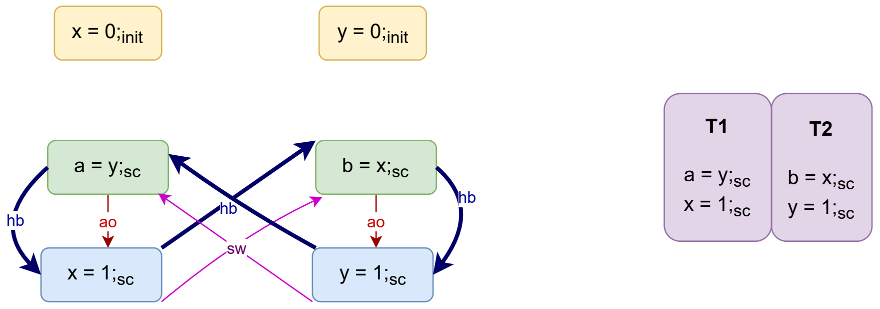


Figure 5.30: Case where a = 1 and b = 1 is creates a happens-before cycle

Observation:

- From the read values we can infer that the Candidate Execution should have $x = 1_{sc} \overrightarrow{hb}\ a = x_{sc}$ and $y = 1_{sc} \overrightarrow{hb}\ a = y_{sc}$.
- The above relations create the cycle $a = y_{sc} \overrightarrow{hb}\ x = 1_{sc} \overrightarrow{hb}\ a = x_{sc} \overrightarrow{hb}\ y = 1_{sc} \overrightarrow{hb}\ a = y_{sc}$.
- This execution is invalid.

A Write e **of type** uo/sc **followed by a Write** d **of type** sc The following example shows a program with a thread having a write of any access mode(uo/sc) followed by a write of type sc.

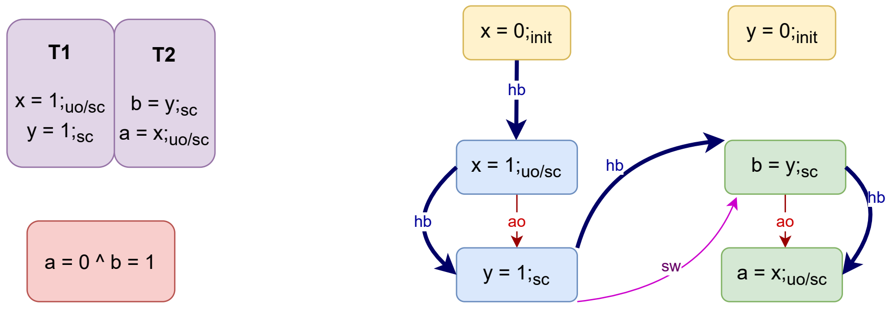


Figure 5.31: Case where a = 0 and b = 1 is invalid due to Coherent Reads.

The figure on the left above shows an example of a candidate where the case of reads in the red box is not possible. The figure on the right shows the Candidate Execution of such a case. Observations:

- From the Candidate Execution, we can infer $x = 0_{init} \xrightarrow[hb]{} x = 1_{uo/sc} \xrightarrow[hb]{} a = x_{uo/sc}$
- This is a pattern that matches the second axiom of Coherent reads, thus restricting the read of a to have the value of 0 as x.
- This inference was due to $y = 1_{sc} \xrightarrow[hb]{} b = y_{sc}$.

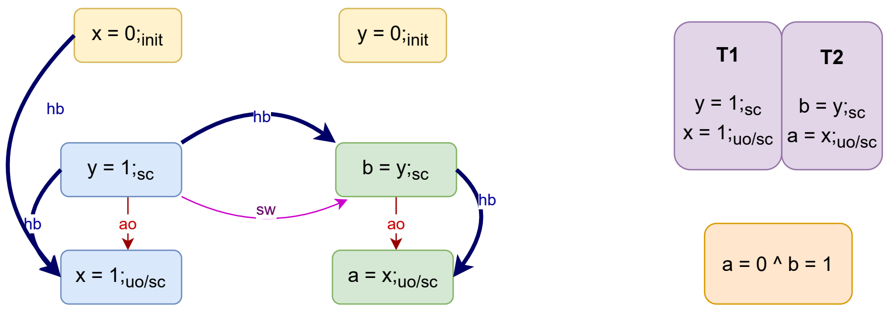


Figure 5.32: Case where events of T1 are reordered, resulting in a = 0 and b = 1 to be valid.

The figure on the right above shows the program after reordering the two events in $T1$ where case of reads in the orange box is possible. The figure on the left shows the Candidate Execution that explains the orange box case. Observations:

- From the Candidate Execution, we can infer $\neg x = 1_{uo/sc} \xrightarrow[hb]{} a = x_{uo/sc}$
- Hence, there is no pattern that the axioms restrict, thus validating x to be read as 0 by a.

All the above counter examples have a lot of repetitive text and can have better formal arguments than a list of observations. Make plans to clean up this once filled in with all the counter examples.

5.3 From Candidates to Program

We first consider programs with conditionals. The following two properties holds for any candidates of programs having conditional branching.

Property 1. *Candidates of Programs with Conditionals (2-branch) Let $B1, B2$ be two sets of events based on each branch of a conditional in a program P. Let C be any Candidate of P, Consider $b1, b2$ to be representative of any event in $B1, B2$ respectively. Then:*

$$\nexists C \in P \ s.t. \ b1 \in C \ \wedge \ b2 \in C$$

There cannot exist any candidate of the program such that events from both sets can be part of it.

Property 2. *Candidates of Programs with Conditionals (1-branch) Let $B1$ be two sets of events based on each branch of a conditional in a program P. Let C be any Candidate of P, Consider $b1$ to be representative of any event in $B1$. Then:*

$$\exists C \in P \ s.t. \ b1 \notin C$$

There exists a candidate of the program such that events from the branch cannot be part of it.

The property for candidates of a program with 1branch conditional may not always be true. Some programs may always satisfy the conditional. From a general point of view, a program represents all possibilities, hence this property suffices.

While the property for 1 branch may not always hold (it can be the case that the branch is always taken in any execution) we are defining it for any program.

Proof. Based on an exeuction of the program, the conditional will either be satisfied or not, but never both. Hence proved both properties. Do we need an elaborate proof of this? As this is direct from existing literature on sequential programs. □

Perhaps we need a general corollary for program level

Corollary 2.1.4. *Reordering under Program with Conditionals Consider a program P and its candidates $C_1, C_2, ..., C_n$ in which events e and d present in all of them with $e \xrightarrow[ao]{} d$. Consider the set of corresponding candidates $C'_1, C'_2, ..., C'_n$ after reordering e and d and its corresponding program P'. Then the set of observable behaviors of P' is a subset of that of P if the following two conditions hold:*

$$Reord(e, d) \ \wedge \ (\forall C_{i \in [1,n]}, \forall k \in C_i \ s.t. \ e \xrightarrow[ao]{} k \wedge k \xrightarrow[ao]{} d, \ Reord(e, k) \wedge Reord(k, d))$$

The above condition can be simplified as we already have corollary to show reordering of non-consecutive events No Candidate of P exists such that only one of e or d exists in them.

$$\nexists C \in P \ s.t. \ (e \in C \ \wedge \ d \notin C) \ \vee \ (e \notin C \ \wedge \ d \in C)$$

Proof. Make the above more formal, and use union of sets and subset property.

We prove the second condition first. Suppose the second condition does not hold. Thus we would have

$$\exists C \in P\ s.t.\ (e \in C\ \wedge\ d \notin C)\ \vee\ (e \notin C\ \wedge\ d \in C)$$

This can only happen, if events e or d are part of a conditional branch.

- C1: Both e and d are part of conditional branches If e and d are in different branches of same conditonal, then by Prop 1, we would have

  $$\nexists C \in P \text{ s.t. } e \in C\ \wedge\ d \in C$$

  But our Corollary assumption is that there exists such candidates. Hence, this cannot be the case.

  If e and d are of the same conditonal branch, then our assumption does not hold by Prop 2, 1.

  if e and d belong to branches of differnt conditionals, then suppose they are part of conditonals of typr Prop 1. Therefore, there exists events $l1, l2$ in their respective counter branches such that:

  $$\nexists C \in P \text{ s.t. } e \in C \wedge l1 \in C$$
  $$\nexists C \in P \text{ s.t. } d \in C \wedge l2 \in C$$

  Reodering e and d would result in program P' such that

  $$\nexists C' \in P' \text{ s.t. } d \in C \wedge l1 \in C$$
  $$\nexists C' \in P' \text{ s.t. } e \in C \wedge l2 \in C$$

  Thus giving us new Candidates in P' not in P such that

  $$e \in C' \wedge l1 \in C'$$
  $$d \in C' \wedge l2 \in C'$$

  Irrespective of e and d being reads or writes, there could be a new $\overrightarrow{rf}$ relation be formed with some event$k$. Thus, we have a new observable behavior.

  From the above we can also conclude that even if e or d (one of them) are in a conditional branch satsifying Prop 1, a new observable behavior can be introduced due to a new Candidate that violates the original Prop 1 of every candidate of program P.

Lastly, suppose both e and d are part of conditonal branches satisfying Prop 2, then we have

$$\exists C \in P \text{ s.t. } e \notin C$$
$$\exists C \in P \text{ s.t. } d \notin C$$

Let B_e and B_d be the respective set of events that belong in the same branch as B_e and B_d respectively. Thus, by Prop 2, we also have

$$e \notin C \;\Rightarrow\; \nexists k \in B_e \text{ s.t. } k \in C$$
$$d \notin C \;\Rightarrow\; \nexists k \in B_d \text{ s.t. } k \in C$$

Now after reordering e and d, we could have a Candidate in P' such that the above conditions are violated. Irrespective of e and d being reads or writes, there could be a new $\overrightarrow{rf}$ relation be formed with some event$k$. Thus, we have a new observable behavior.

Think about this later, discuss with Clark as to how to explain this better. Use diagrams perhaps. This is not well justified. Perhaps place the above condition as a property of each conditional. It would be easier to reason with.

One may think of it as introducing a new event in a Candidate, thus causing new observable behavior.

- C2: Without loss of generality, let us consider e is part of conditional but d is not.

  By Prop 1, there would exist some event l in another branch such that

  $$\nexists C \in P \text{ s.t. } e \in C \;\wedge\; l \in C$$

  On reodering e and d, we have

  $$\nexists C \in P' \text{ s.t. } d \in C \;\wedge\; l \in C$$

  Thus we have

  $$\exists C \in P' \text{ s.t } e \in C \;\wedge\; l \in C$$

  giving us a new Candidate in P' not in P. Irrespective of e being a read or a write, there could be a new $\overrightarrow{rf}$ relation be formed with some event$k$. Thus, we have a new observable behavior. By Prop 2, we would have

  $$\exists C \in P \text{ s.t. } e \notin C$$

On reodering e and d, we have

$$\exists C \in P' \text{ s.t. } d \notin C$$

Thus we have

$$\nexists C \in P' \text{ s.t } e \notin C$$

giving us a new Candidate in P' not in P. Irrespective of e being a read or a write, there could be a new $\xrightarrow[rf]{}$ relation be formed with some event$k$. Thus, we have a new observable behavior.

Finding no such event k would involve gathering more information about other Agent events. Moreover,obtaining such information may be infeasible as the Compiler must ensure that there is No candidate which has such anevent k given that there is event e as per our condition. Thus, conservatively we can consider doing suchreordering to be unsafe.

Now that we have that the second condition must hold, we prove the first condition too must hold. Let C_i and C_i' be the candidates before and after reordering e and d. From the first condition we have then for C_i

$$\forall\ k\ s.t.\ e \xrightarrow[ao]{} k\ \wedge\ k \xrightarrow[ao]{} d\ .\ Reord(e,k)\ \wedge\ Reord(k,d).$$

The above is Corollary 1 (tag properly), thus giving us that the observable behaviors of C_i' is a subset of C_i. By property of unions of sets, we can conclude that the set of Observable Behaviors of P' is a subset of that of P.

Hence proved.

Perhaps we need to define a function Obs, that takes a candidate and gives us a set of rf / rbf relations. THat would help shaping the argument of observable behavior subset well written. □

It has come to my notice that in general reordering may not be fine, if we end up removing something outside a conditional. To show this I have a simple counter example. However, the only way to show this, is to say that there is some candidate which suddenly has an agent order relation between two events which were not supposed to have any relation to them. Simply put, we can say that there is a candidate execution of the reordered program, where both the events exist, where as there isn't any candidate of the original program where such a thing can happen.

Next, we consider programs with loops.

Programs with loops might have a little difficulty in defining constraints on Candidates. This is because if we couple them with conditionals then it may be that one iteration would have some events of the loop while the other set of iterations will not have. How then can we define the general case? Perhaps I can consider only programs with loops but no conditionals, then later consider programs having both.

- A :
- B
- C
- D
- E

5.3.1

Chapter 6

Elimination

6.1 Elimination

There are two types of elimination we are concerned with:

- Read Elimination
- Write Elimination

Theorem 2.2. *Consider a candidate C of a program and its possible Candidate Executions where $\xrightarrow[hb]{}$ is strictly partial order. Consider an event e which is a read. Consider another Candidate C' without the event e. If e has an unordered access mode, then the set of Observable behaviors of C' is a subset of C without the relation $e \xrightarrow[rf]{} w$ where w is some write event in C.*

Proof. We look at this as an elimination of e that takes place in any candidate execution of C. We then go about answering the same four questions as we did for reordering. The only major change here being that elimination removes $\xrightarrow[hb]{}$ relations. We must check whether the removal of these relations introduce new behaviors, in contrast to that in reordering, where new relations were introduced.

1. Preserving *happens-before* relations The relations we want to preserve are those that are dervied through relation with e, meaning the following two relations:

a) $k \xrightarrow[hb]{} e$ b) $e \xrightarrow[hb]{} k$

We can divide the events involved in the above into two sets:

$$K_b = \{k \mid k \xrightarrow[hb]{} e\}.$$
$$K_a = \{k \mid e \xrightarrow[hb]{} k\}.$$

Put a figure here for an intuitive understanding of the problem at hand

We need to ensure the following relations hold after elimination.

$$\forall k_a \in K_a \ \wedge \ \forall k_b \in K_b \ . \ k_b \xrightarrow[hb]{} k_a \tag{6.1}$$

Slight notational confusion WHat if the eliminated event is a conditional check? That would mean events in the conditional check are also eliminated. Which would mean one has to check if it is okay to eliminate all events within the conditional.

Similar to reordering, we need to have a valid pivot pair $< p_b, p_a >$ such that

$$\forall k_b \neq p_b \in K_b \ . \ k_b \xrightarrow[hb]{} p_b \tag{6.2}$$

$$\forall k_a \neq p_a \in K_a \ . \ p_a \xrightarrow[hb]{} k_a \tag{6.3}$$

By Lemma 1, $e : uo$ is the only condition that satisfies our requirement. By Lemma 2, $e : uo \ \vee \ e : sc$ are the options. Condsidering both the above conditions to be satisfied, $e : uo$ is the only possibility that holds.

Write an expression which is the conjunction of both lemmas, and show how the conjunction boils down to the result that we come to.

2. The *happens-before* relations lost The relations lost are those attached to the event e, which are:

$$k \xrightarrow[hb]{} e \ \vee \ e \xrightarrow[hb]{} k \tag{6.4}$$

Do we need to prove that these are the only relations lost? Proof part 1 implicitly shows this.

3. Presence of Cycles? Because no new $\xrightarrow[hb]{}$ relations are introduced, and because original candidate executions have $\xrightarrow[hb]{}$ as a strict partial order, no cycles are introduced after elimination.

Perhaps write this argument a bit better.

4. Do the lost relations result in New Observable Behaviors? To answer this, we need to see whether the relations removed had an impact on $\xrightarrow[rf]{}$ relations other than those with e. To prove that it does not have any impact, we divide our argument into two parts, viz. into the two types of relations removed:

a) $k \xrightarrow[hb]{} R_u o$

b) $R_u o \xrightarrow[hb]{} k$

In the first case, we have the following possibilities.

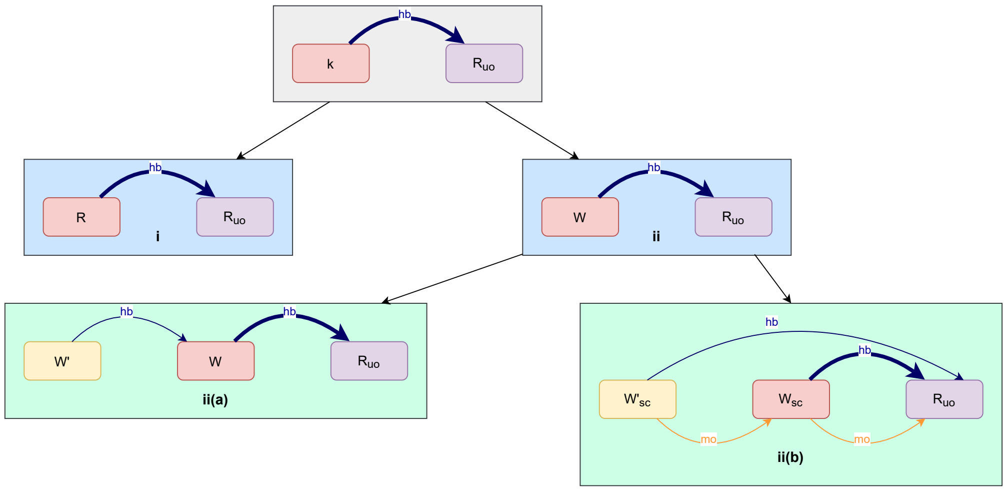


Figure 6.1: The first type of relations removed and the various patterns forbidden by them.

Observations:

- (i) is not a pattern forbidden by the consistency rules
- (ii)(a) is a pattern in Coherent Reads, however, only restricting $\overrightarrow{rf}$ relation with R and W'(which here is our Unordered Read)
- (ii)(b) is a pattern in Sequentially Consistent Atomics, however, once again only restricting $\overrightarrow{rf}$ relation with R and W'.

In the second case, we have the following possibilites.

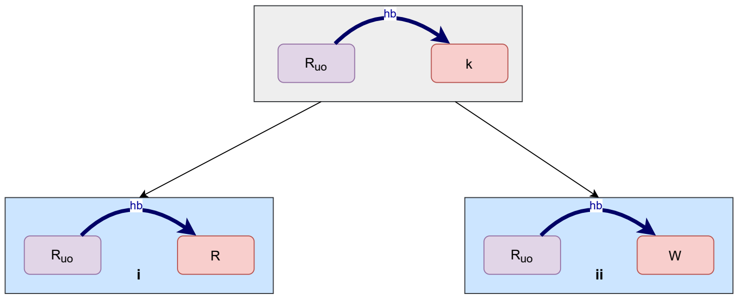


Figure 6.2: The second type of relations removed and the various patterns forbidden by them.

Observations:

- (i) is not a pattern in any Consistency rules
- (ii) is a pattern in Coherent Reads, however, only restricting $\xrightarrow[rf]{}$ relation with R and W

From the above observations, we can see that the relations removed only have restriction on reads-from relations on the event we elmiinate. Thus, by case wise analysis we can conclude that no new observable behaviors are introduced due to the removed $\xrightarrow[hb]{}$ relations.

□

Explain here why we consider two consecutive write events only. The argument being the Coherent Reads pattern can be triggered anyhow.

Theorem 2.3. *Consider a candidate C of a program and its possible Candidate Executions where $\xrightarrow[hb]{}$ is strictly partial order. Consider two* ***write*** *events e and d such that $cons(e, d)$ is true in C and $e \xrightarrow[ao]{} d$. Consider a Candidate C' without event e. If e has an unordered access mode and e and d have the same range, then the set of Observable behaviors of C' is a subset of C.*

Proof. Once again, we look at this as a write elimination done on a Candidate Execution of C. We start by proving when other happens-before relations remain intact. Followed by identifying relations lost due to elimination and a proof for when these relations do not introduce new observable behaviors.

Preserving Happens-before relations The relations we want to preserve are those that are dervied through relation with e, meaning the following two relations:

a) $k \xrightarrow[hb]{} e$ b) $e \xrightarrow[hb]{} k$

We can divide the events involved in the above into two sets:

$$K_b = \{k \mid k \xrightarrow[hb]{} e\}.$$
$$K_a = \{k \mid e \xrightarrow[hb]{} k\}.$$

Put a figure here for an intuitive understanding of the problem at hand

We need to ensure the following relations hold after elimination.

$$\forall k_a \in K_a \ \wedge \ \forall k_b \in K_b \ . \ k_b \xrightarrow[hb]{} k_a \tag{6.5}$$

Slight notational confusion

Similar to reordering, we need to have a valid pivot pair $< p_b, p_a >$ such that

$$\forall k_b \neq p_b \in K_b \ . \ k_b \xrightarrow[hb]{} p_b \tag{6.6}$$
$$\forall k_a \neq p_a \in K_a \ . \ p_a \xrightarrow[hb]{} k_a \tag{6.7}$$

By Lemma 1, $e\!:\!uo$ is the only condition that satisfies our requirement. It can be our p_a and by Lemma 2, $e\!:\!uo \ \vee \ e\!:\!sc$ are the possibilities. Condsidering both the above conditions to be satisfied, $e\!:\!uo$ is the only possibility that holds.

Again, show the conjuction of both conditions

2. The *happens-before* relations lost The relations lost are those attached to the event e, which are:

$$k \xrightarrow[hb]{} e \ \vee \ e \xrightarrow[hb]{} k \tag{6.8}$$

Do we need to prove that these are the only relations lost? Proof part 1 implicitly shows this.

3. Presence of Cycles? Because no new $\xrightarrow[hb]{}$ relations are introduced, and because original candidate executions have $\xrightarrow[hb]{}$ as a strict partial order, no cycles are introduced after elimination.

Perhaps write this argument a bit better.

4. Do the lost relations result in New Observable Behaviors? To address this, we divide our cases into two parts; one for each type of relation lost:

a) $k \xrightarrow[hb]{} e$ b) $e \xrightarrow[hb]{} k$

For the first case, we have the following possibilities:

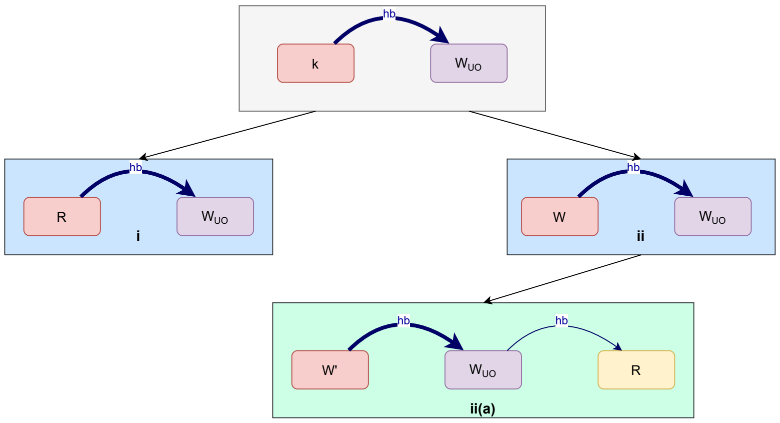


Figure 6.3: First case possibilities (change caption stimiar to that for read elim)

We can observe the following:

- (i) is a pattern from Coherent Reads that restricts the read R reading from W. And this will remain the case even after elimination of W.
- (ii)(a) is a pattern from Coherent reads, forbidding R to read from some W'. This will remain the case after elimination of W if firstly we have $d \xrightarrow[hb]{} R$. By Lemma 2 this is indeed the case. Secondly, we need to ensure that after elimination, the Coherent Reads pattern with d now restricts the exact set of $\xrightarrow[rbf]{}$ relations. Since we have no certain information on the range of R or W', we require the ranges of e and d to be same for our requirement to hold in general.
- **PErhaps explain the above argument in more detail**

For the first case, we have the following possibilities:

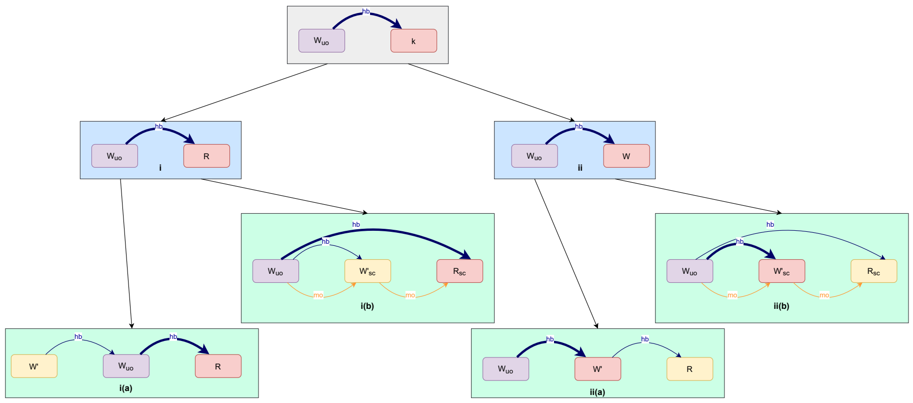


Figure 6.4: Second case possibilities (change caption stimiar to that for read elim)

We make the following observations:

- (i)(a) has the similar argument to the previous case's (ii)(a), requiring e and d to have equal ranges.
- (i)(b) is a pattern of Sequentially Consistent Atomics, which restricts R from reading anything of W. This will reamin the case after W is eliminated.
- (ii)(a) is a pattern of Coherent Reads, restricting R from reading W. This will remain the case after eliminating W.
- (ii)(b) is the same as (i)(b), hence the argument remains the same.

In all the above cases, observe that on keeping range of e and d equal, none of the patterns introduce any new observable behavior. Hence, if we have two consecutive writes of equal ranges, of which the first one has access mode unorderd, the set of Observable Behaviors without the write is a subset of that with it present.

$\square$

Corollary 2.3.1. *Consider a Candidate C of a program and its Candidate Executions which are valid. Consider two events e and d both having equal ranges such that:*

$$e \in W \ \wedge \ d \in W \ \wedge \ e : uo \ \wedge \ e \xrightarrow[ao]{} d \ \wedge \ \neg cons(e, d)$$

Consider another Candidate C' without the event e. If

$$\forall k \ s.t. \ e \xrightarrow[ao]{} k \wedge k \xrightarrow[ao]{} d \ , \ Reord(e, k)$$

Then, the set of Observable behaviors possible in C' is a subset of C.

Proof. We prove by induction on the number of events k between e and d. We verify that if a j exists that is valid, the Observable behaviors of C' is a subset of C.

Base Case : n = 1 We have the case when:

$$e \xrightarrow[ao]{} k_1 \;\wedge\; relnk_1aod$$

By Theorem of Reordering and Def of consecutive events and agent order, we can reorder e and k_1, thus giving us a Candidate C'' with :

$$k_1 \xrightarrow[ao]{} e \;\wedge\; e \xrightarrow[ao]{} d$$

whose observable behaviors are a subset of C.

By Def of Consecutive instructions and Theorem of Elmination, we can eliminate e, thus giving us candidate C' with

$$k_1 \xrightarrow[ao]{} d$$

whose observable behaviors are a subset of C''.

By transitive property of subsets, we can conclude that the observable behaviors of C' is a subset of C.

Inductive Case (n) Let us assume that if the number of events in between are n, then the corollary holds. Let us consider the Candidate to be C_n and corresponding candidate after elimination as C'_n. The observable behavior of C'_n is a subset of that of C_n.

If we can show the above holds true for $n+1$ events, we are done.

To show this, suppose we have C_{n+1} as the candidate and C' as the one after elimination of e.

Because $\xrightarrow[ao]{}$ is a total order, there is a total order among all $n+1$ events k agent ordered between e and d such that we can label them $k_1, k_2, ..., k_{n+1}$ with the following properties

$$e \xrightarrow[ao]{} k_1 \xrightarrow[ao]{} ... \xrightarrow[ao]{} k_{n+1} \xrightarrow[ao]{} d \;\wedge\; cons(e, k_1) \wedge cons(k_1, k_2) \wedge ... \wedge cons(k_{n+1}, d)$$

By Theorem of Reordering and Def of consecutive events and agent order, we can reorder e and k_1, thus giving a corresponding candidate C_n having observable behaviors as a subset of C_{n+1}.

By our inductive assumption, we have that the observable behaviors of C' is a subset of C_n. By transitive property of subsets, we can then conclude that the observable behaviors of C' are a subset of that of C_{n+1}.

□

The above proof is clear, but it seems to me that I need to label all definitions and lemmas and theorems and corollary so that I can refer them here.

6.2 From Candidates to Program

Corollary 2.3.2. *Consider a program P and its candidates $C_1, C_2, ..., C_n$ in which events e and d present such that*

$$e \in W \;\; \wedge \;\; d \in W \;\; \wedge \;\; e : uo \;\; \wedge \;\; e \xrightarrow[ao]{} d \;\; \wedge \;\; \Re(e) = \Re(d)$$

. Consider the set of corresponding candidates $C'_1, C'_2, ..., C'_n$ after eliminating e and its corresponding program P'. If

$$\forall C_{i \in [1,n]}, \forall k \in C_i \;\; s.t. \;\; e \xrightarrow[ao]{} k \wedge k \xrightarrow[ao]{} d, \;\; Reord(e, k)$$

and

$$\nexists C \in P \;\; s.t. \;\; e \in C \wedge d \notin C$$

Then the set of observable behaviors of P' is a subset of that of P.

Proof. We first prove that the second condition must hold. We show this by proving that if it does not hold, a new observable behavior can be introduced.

Suppose the second condition does not hold, then we have

$$\exists C \in P \text{ s.t. } e \in C \wedge d \notin C$$

By Prop 2 and Prop 2, we can infer that the above holds if e or d are part of a conditional branch.

- Case 1: e and d both are part of conditionals By Prop 1 and 2, we have

$$\exists C \in P \text{ s.t. } d \notin C$$
$$\exists C \in P \text{ s.t. } e \notin C$$

  After elimination e, we can have a new observable behavior in a candidate not having d as above condition states.

  Need to refer to part of elimination proof as Coherent Reads would not be triggered anymore for a case and thus we can have a new observable behavior. How to explain this, ask Clark.

- Case 2: e is part of conditional but d is not By Prop 1 and 2, we have

  $$\exists C \in P \text{ s.t. } e \notin C$$

  After elimination e, we cannot have a new observable behavior in a candidate due to not having d as above condition states.

- Case 3: d is part of a conditional but e is not

  By Prop 1 and 2, we have

  $$\exists C \in P \text{ s.t. } d \notin C$$

  After elimination e, we can have a new observable behavior in a candidate not having d as above condition states.

  Need to refer to part of elimination proof as Coherent Reads would not be triggered anymore for a case and thus we can have a new observable behavior. How to explain this, ask Clark.

  Add the above property to conditionals with two branches also.

Now that we have that the second condition must hold, we prove the first condition too must hold. Let C_i and C_i' be the candidates before and after eliminating e. From the first condition we have then for C_i

$$\forall\ k\ s.t.\ e \xrightarrow[ao]{} k\ \wedge\ k \xrightarrow[ao]{} d\ .\ Reord(e, k).$$

The above is Corollary 1 (tag properly) for elimination, thus giving us that the observable behaviors of C_i' is a subset of C_i. Hence this condition must hold for all candidates from which we eliminate e.

By property of unions of sets, we can conclude that the set of Observable Behaviors of P' is a subset of that of P.

Hence proved.

We have not given properly the link between Observable Behaviors, Candidate Executions, Candidates and Programs. Perhaps we need to define a function Obs that gives us the set of Observable Behaviors, where the Domain can be a Program, Candidate, or Candidate Execution.

□

As far as read elimination goes, since we only need the information of read event that is to be eliminated, we do not need to take cases as above for write elimination. Except there can exist one case, in which the read itself is the conditional check, which can result in choosing any branch based on the conditional being satisfied or not.

The compiler can ideally choose to eliminate the read event, if the choice of branch has not effect on the execution of the thread. However, the events in each branch can affect the execution of other threads and thus have a role in the possible observable behaviors. But if all the events in each branch are eliminable, then we can safely assume that it does not matter which branch is taken, as although the set of observable behaviors would be different, they would result in the same candidate after events in the chosen branch are eliminated. THis is what we assume the compiler intends to do by eliminating the conditional.

Explain this a little better.

Corollary 2.3.3. *If the read to be eliminated is a conditional, then only if all events within the conditional can be eliminated, will this be safe to do. The problem is that this is recursive, and hence the proof must also have some sort of recursion. But to be frank, the proof would not have any assumption on the structure of flow of code within each conditional branch, just that we can eliminate some event and what does that imply in terms of observable behaviors.*

In order to show we can eliminate a write, we would need some other write po after and that respects the corollary before this one. Perhaps mention that the compiler needs to do a flow analysis to get this done. And this is not that arduous a task in terms of algorithmic complexity.

Proof. By thoerem of reordering, we can only eliminate the read when it is of type unordered.

To prove the next part, consider two candidates C_1 and C_2, one for each branch taken. Our task is to show that the observable behavior of both these candidates should be the same after the removal of read, which in our terms means the removal of each branch.

But is it so? The compiler can choose to elimiate the conditional because it always turns out to be only true or only false.

If it is a write, then we need to have some other write program ordered ahead to show that we can eliminate it. Perhaps put this as a general conditioon and write a boolean funciton to state when a write is eliminable.

If it is a read, then we only require it to have access mode unordered.

Perhaps we indeed need more information as to why the compiler is eliminating the conditional check.

- The read removed is the shared memory one.
- Even if the read is of type unordered, removing it will make the branch choice uncertain.
- If the compiler thinks that only one branch will always be taken, then it could safely eliminate.

- But we place no assumption on why the compiler is doing so, hence in our eyes, the choice of branch is uncertain.
- Given that the compiler does do it with the assurance that the conditional check is futile, one may elimiate the read.
- Perhaps a proof by counter example would suffice to show whether it is safe to do so.
- One best way is to observe that all events that are supposed to happen in each conditional branch, happens nevertheless after the branches merge. So one might want to eliminate events.
- To do the above, the compiler must assure that it can eliminate all events in each branch. If it can do so, then elimianting the read will be safe.
- The above can be shown to hold as it is a simple argument based on Observable Behaviors, union of sets and transitive property of subsets.
- However, the question remains as to whether the compiler thinks the shared memory conditional check read is alwys the same value or whether it considers the branch choice irrelevant for the thread.
- If we say that we can eliminate all events in each branch, it shows that these events may not be needed. But if cannot eliminate all events in the branches, it may not imply that it is still safe to eliminate. Discuss with Clark.

$\square$

Chapter 7

Our Critique of The Model

Chapter 8

Conclusion, Summary, Future Work