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

Abstract

Abrégé

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Concurrent programs take advantage of *out-of-order* execution. Intuitively, this means that more than one unrelated computations can be done "simultaneously" without having any fixed order in which they should happen. This results in concurrent programs having multiple different outcomes, the possible outcomes of which are described by a *memory consistency model*. The most intuitive and commonly relied upon model is that of *Sequential Consistency* (SC), which guarantees that every outcome of a program must be equivalent to a sequential interleaving of each thread's individual actions. For example, consider the program in Figure **??** with two threads, which share memory denoted by x, y initialized to 0, where a, b are local variables. The right-hand-side are the possible values that a and b can read under sequential consistency rules.

Figure 1.1: Example program with its possible outcomes under sequential consistency.

However, the above program under SC cannot have the outcome $a = 2 \ \wedge \ y = 0$. From a program transformation standpoint, such an outcome should be possible: we can simply reorder either both the reads or both writes to x and y, as they are computations on disjoint memory. But from a consistency rule standpoint, since the

outcome is not valid, it also brings with it the conclusion that such simple program transformations may not be safe or even invalid. Figure **??** shows how after doing either one of these reorderings, an outcome invalid under SC is possible.

Figure 1.2: Left program is when the two reads in T1 are reordered, whereas the right program is when the two writes of T2 are reordered.

Yet another example can be that of elimination. (show an example where elimination could result in problems. Refer your presentation.)

Weaker consistency models have been introduced to concurrent, shared-memory languages to leverage more of the *out-of-order* notion. For instance, under the ECMAScript consistency model semantics, if all the accesses are of type *unordered*, the above invalid outcome is allowed, which implies a reordering of such events is valid in the above case. The problem though is that semantics of such weak consistency can be easily misunderstood, and is sometimes defined in informal prose format, thus leading to misinterpretation of intended semantics, which leads to implementation issues. The lack of clear semantics also makes it difficult to assert when a particular program transformation is valid / safe (in our case, instruction reordering).

Our focus in this thesis is to offer a clarified, more concise rendition of the core ECMAScript memory model that allows for better abstract reasoning over allowed and disallowed behaviours (outcomes). We use our model to provide a straightforward, conservative proof of when reordering and elimination of independent instructions is permitted, addressing optimization in terms of its impact on observable program behaviours. Specific contributions of our work include the following:

1. We provide a concise *declarative style* model of the core ECMAScript memory consistency semantics. This clarifies the existing draft presentation [**?**] in a manner useful for validating optimizations.

2. Using our model we show when basic reordering of independent instructions is allowed. Although conservative, this represents a formal proof when the fundamental optimization is permitted.

3. Similar proof designs are used to validate other basic optimization behaviours such as removing redundant reads or writes.

Chapter 2

Background

Chapter 3

Related Work

Chapter 4

The Memory Model

This chapter starts with exposing some problems with the existing specifications of the model. The latter part introduces the key components of the model that we find essential to address program transformations. We start by introducing Agents and Event sets, followed by various Binary Relations defined between events. We then introduce certain helper definitions that prove useful in understanding the Axioms of the model. We then use the binary relations and definitions to specify the Axioms of the model. Lastly, we define Races followed by defining what a Consistent Execution is as per the specificaiton of the model.

The model we consider is the current draft specification (cite here) of ECMAScript standard. This draft as far as the memory model goes has remain unchanged, so we believe our work will also be of use to those working on thismodel. The specification is claimed to be axiomatic by definition, which should, in our view remove the complexities of the rest of thestandard from the semantics of the model. However, as we noted, there were quite some concerns with it:

The Model is Quite Algorithmic Although the standard states that the model is not supposed to be operational, the specificaitons of the model state otherwise. There are quite a few abstract operations which are not necessary to understand the semantics of the model. As an example, consider one of the Axioms of the model below as stated by the standard.

28.7.2 Coherent Reads

A candidate execution *execution* has coherent reads if the following abstract operation returns **true**.

1. For each ReadSharedMemory or ReadModifyWriteSharedMemory event *R* of SharedDataBlockEventSet(*execution*), do
    a. Let *Ws* be *execution*.[[ReadsBytesFrom]](*R*).
    b. Let *byteLocation* be *R*.[[ByteIndex]].
    c. For each element *W* of *Ws*, do
        i. If (*R*, *W*) is in *execution*.[[HappensBefore]], then
            1. Return **false**.
        ii. If there is a WriteSharedMemory or ReadModifyWriteSharedMemory event *V* that has *byteLocation* in its rang
        such that the pairs (*W*, *V*) and (*V*, *R*) are in *execution*.[[HappensBefore]], then
            1. Return **false**.
        iii. Set *byteLocation* to *byteLocation* + 1.
2. Return **true**.

Figure 4.1: The definition for Coherent Reads

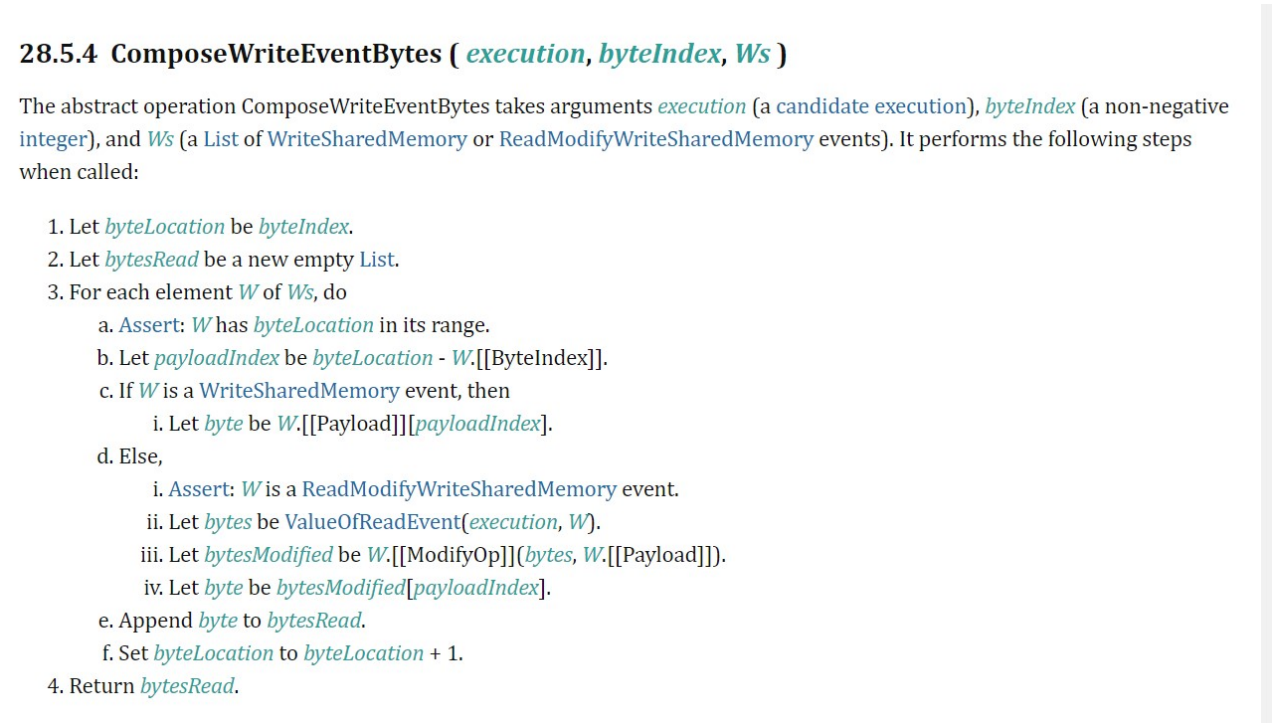
The above axiom is specified as a return value to an abstract operation. While understanding this requires one to know thedefintions of Ws, $execution$, $SharedDataBlockEventSet$ abstract operation, etc. we believe this is not needed as to understandwhat the axiom is about, which informally can be stated as below in two points:

- A read's value cannot come from a write that has happened after it.
- A read's value cannot come from a write that has been overwritten by some other write.

Axiomatically, we define the above two points using simple binary relations that

we derive from the specification to exist amongevents. The entire specification is structured in a similar way.

Certain Unnecessary Definitions Certain abstract operations are not required to capture the semantics of the model. One such example is in the figure below:



28.5.4 ComposeWriteEventBytes (*execution*, *byteIndex*, *Ws*)

The abstract operation ComposeWriteEventBytes takes arguments *execution* (a candidate execution), *byteIndex* (a non-negative integer), and *Ws* (a List of WriteSharedMemory or ReadModifyWriteSharedMemory events). It performs the following steps when called:

1. Let *byteLocation* be *byteIndex*.
2. Let *bytesRead* be a new empty List.
3. For each element *W* of *Ws*, do
    a. Assert: *W* has *byteLocation* in its range.
    b. Let *payloadIndex* be *byteLocation* - *W*.[[ByteIndex]].
    c. If *W* is a WriteSharedMemory event, then
        i. Let *byte* be *W*.[[Payload]][*payloadIndex*].
    d. Else,
        i. Assert: *W* is a ReadModifyWriteSharedMemory event.
        ii. Let *bytes* be ValueOfReadEvent(*execution*, *W*).
        iii. Let *bytesModified* be *W*.[[ModifyOp]](*bytes*, *W*.[[Payload]]).
        iv. Let *byte* be *bytesModified*[*payloadIndex*].
    e. Append *byte* to *bytesRead*.
    f. Set *byteLocation* to *byteLocation* + 1.
4. Return *bytesRead*.

Figure 4.2: The definition for Compose Write Event Bytes

The above figure is the definition of an abstract operation. To understand what this operation does, one must know the meaning ofthe terms $ModifyOp$, $Payload$, the list Ws, and also know what the argument $ByteIndex$ signifies. In its essence, the above operation gives the read-values read by a single write by collecting the values

from their correspondingwrites. We realized that one need not know this operation nor understand its function as it does not play a role in the semantics of themodel. Other such abstract operations are $ValueOfReadEvent$ and $ValidChosenReads$.

Still a bit verbose The entire model, is still quite verbose, which makes it difficult to understand the main objective of the model semantics. The following figure is the std specification of another Axiom

28.7.4 Sequentially Consistent Atomics

For a candidate execution *execution*, memory-order is a strict total order of all events in EventSet(*execution*) that satisfies the following.

- For each pair (*E*, *D*) in *execution*.[[HappensBefore]], (*E*, *D*) is in memory-order.
- For each pair (*R*, *W*) in *execution*.[[ReadsFrom]], there is no WriteSharedMemory or ReadModifyWriteSharedMemory event *V* in SharedDataBlockEventSet(*execution*) such that *V*.[[Order]] is SeqCst, the pairs (*W*, *V*) and (*V*, *R*) are in memory-order, and any of the following conditions are true.
  - The pair (*W*, *R*) is in *execution*.[[SynchronizesWith]], and *V* and *R* have equal ranges.
  - The pairs (*W*, *R*) and (*V*, *R*) are in *execution*.[[HappensBefore]], *W*.[[Order]] is SeqCst, and *W* and *V* have equal ranges.
  - The pairs (*W*, *R*) and (*W*, *V*) are in *execution*.[[HappensBefore]], *R*.[[Order]] is SeqCst, and *V* and *R* have equal ranges.

  NOTE 1 This clause additionally constrains SeqCst events on equal ranges.
- For each WriteSharedMemory or ReadModifyWriteSharedMemory event *W* in SharedDataBlockEventSet(*execution*), if *W*.[[Order]] is SeqCst, then it is not the case that there is an infinite number of ReadSharedMemory or ReadModifyWriteSharedMemory events in SharedDataBlockEventSet(*execution*) with equal range that is memory-order before *W*.

  NOTE 2 This clause together with the forward progress guarantee on agents ensure the liveness condition that SeqCst writes become visible to SeqCst reads with equal range in finite time.

A candidate execution has sequentially consistent atomics if a memory-order exists.

Figure 4.3: The Axiom of Sequentially Consistent Atomics

The above figure, though written concisely, in our view still makes it difficult to understand. We reduced the above entire axiom into three main patterns using

binary relations that should not exist in any execution of aprogram. While the latter part is less of a semantic specification and more of a programming guideline while using relaxed memory accesses.(one can make countless counter-examples for this.)

Given the above concerns about the model specificaiton in the standard, we formalized it axiomatically in the form of binary relationsover events involved and axioms that place restrictions on certain binary relations using the others.

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.

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:

[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.

- **Shared Memory (*SM*) Events**

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

- **Synchronize (*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].

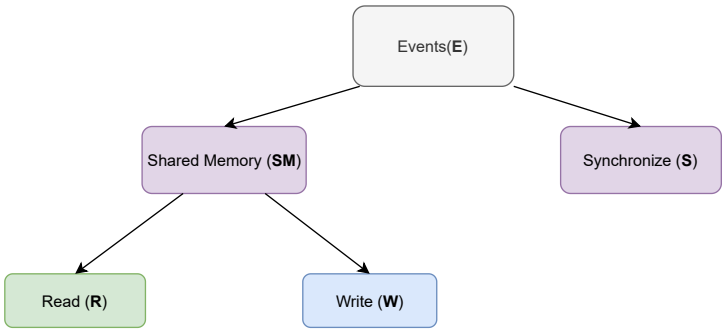


Figure 4.4: The hierarchical categories of different sets of events.

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

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

[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.

[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.

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.

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: $event_{type}$.

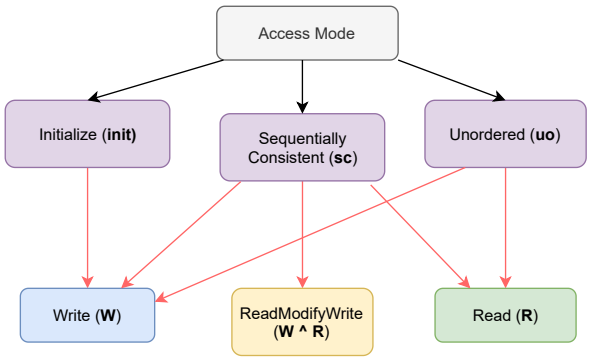


Figure 4.5: Access Modes

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]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.

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 ($\overrightarrow{rbf}$) 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 $\overrightarrow{rbf}$ relation could be represented as

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

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 ($\overrightarrow{rf}$) 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.

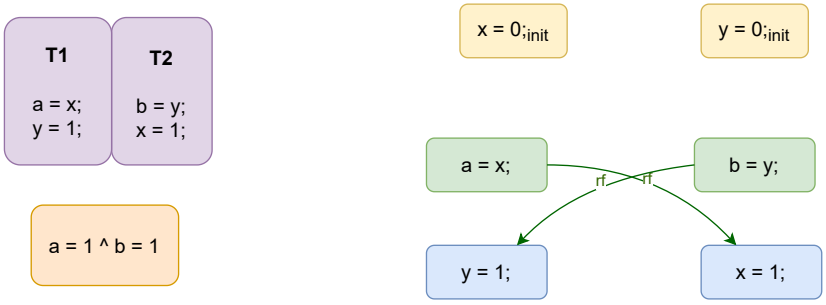


Figure 4.6: An example to show the reads-from relations that are drawn for the example program between read and write events.

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.

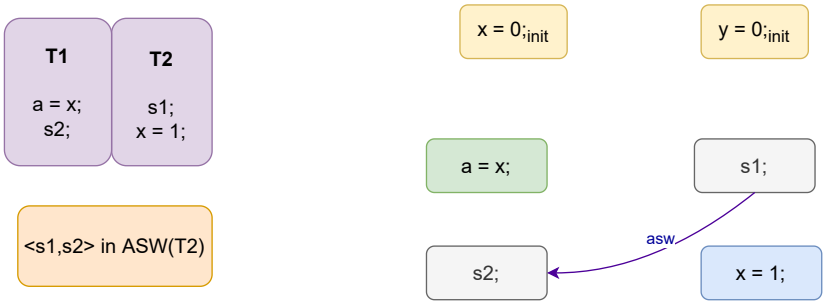


Figure 4.7: An example to show the reads-from relations that are drawn for the example program between read and write events.

4.4 Ordering Relations among Events

4.4.1 Agent Order ($\overrightarrow{ao}$)

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 $e \overrightarrow{ao} d$ or $d \overrightarrow{ao} e$.

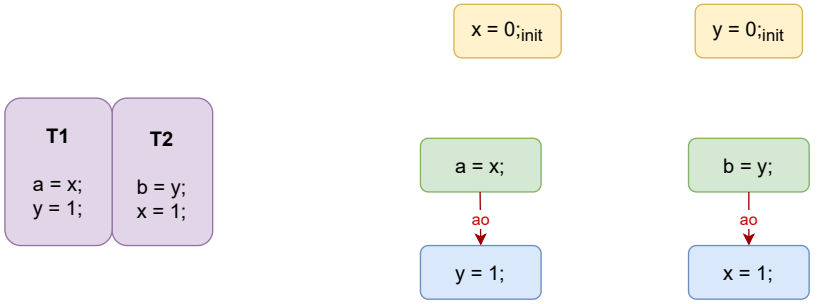


Figure 4.8: An example with agent order among the events.

[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.

4.4.2 Synchronize-With Order ($\overrightarrow{sw}$)

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

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

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

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

$$(r \overrightarrow{rf} w) \ \wedge \ r : sc \ \wedge \ w : sc \ \wedge \ (\Re(r) = \Re(w)) \ \Rightarrow \ (w \overrightarrow{sw} r)$$

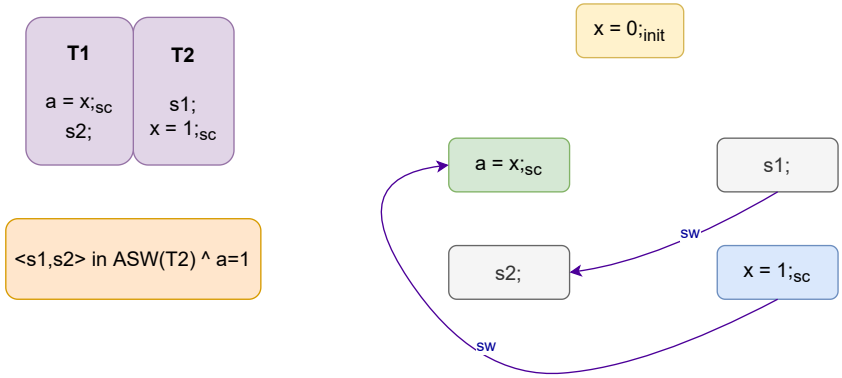


Figure 4.9: An example with synchronize with relations among the events.

4.4.3 Happens Before Order ($\overrightarrow{hb}$)

A transitive order on events, composed of the following:

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

$$(e \overrightarrow{ao} d) \ \Rightarrow \ (e \overrightarrow{hb} d)$$

[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.)

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

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

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

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

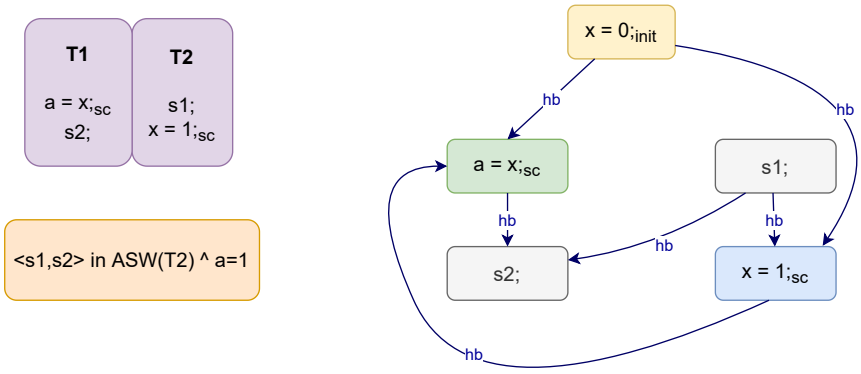


Figure 4.10: An example with all the types of happens-before relations between events.

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

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

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

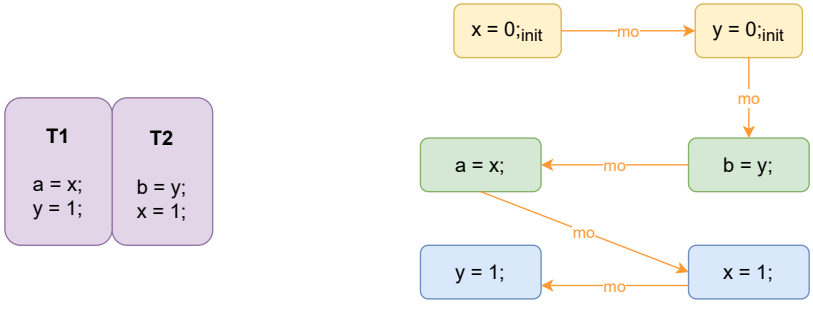


Figure 4.11: An example with a memory order (total) among all events.

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 Helper Definitions

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 4.5.1. *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 4.5.2. *Candidate A a collection of abstracted set of shared memory events of a program involved in one possible execution, with the* $\overrightarrow{ao}$ *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 4.12.*

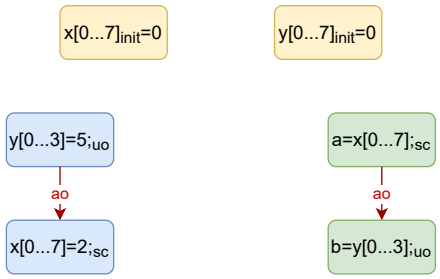


Figure 4.12: An example of a Candidate

Definition 4.5.3. *Candidate Execution A Candidate with the addition of* $\overrightarrow{sw}$, $\overrightarrow{hb}$ *and* $\overrightarrow{mo}$ *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.*

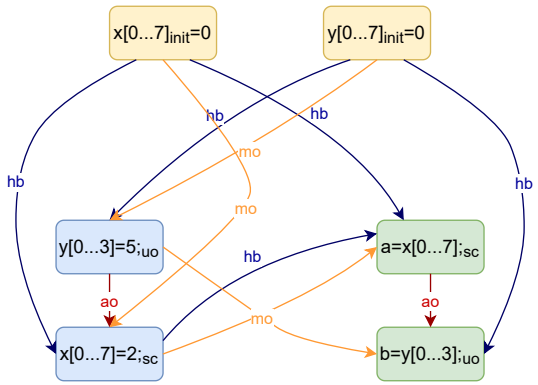


Figure 4.13: An example of an Execution based on Candidate above

Definition 4.5.4. *Observable Behavior*

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

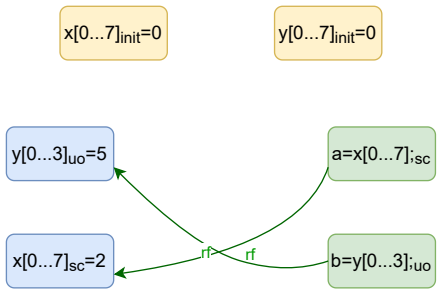


Figure 4.14: Observable Behavior

Definition 4.5.5. *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 $CE_1, CE_2, ...CE_m$[8]. Then, we have the following properties that hold:

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

[8]Note that the variables n and m need not be finite.

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.

Axiom 1. *Coherent Reads*

There are certain restrictions of what a read event cannot see at different points of execution based on $\overrightarrow{hb}$ *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*

$$e \ \overrightarrow{hb} \ d \ \Rightarrow \ \neg \ e \ \overrightarrow{rf} \ d.$$

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

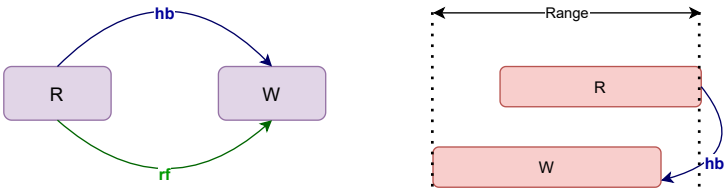


Figure 4.15: A read value cannot come from a write that has happened after it

- *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.*

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

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

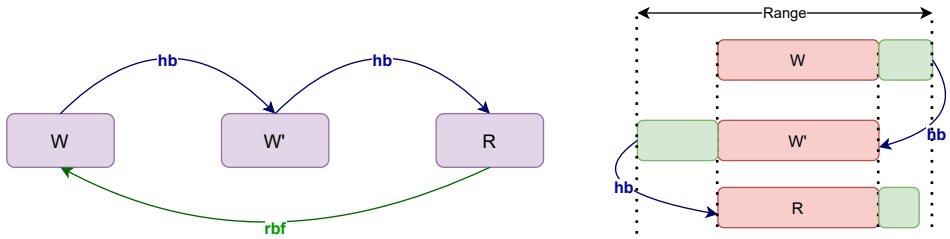


Figure 4.16: A read value cannot come from a write if there is a write that happens between them, writing to the same memory:

Axiom 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[9].

$$d{:}tf \ \wedge \ g{:}tf \ \wedge \ e{:}tf \ \wedge \ (\Re(d){=}\Re(g){=}\Re(e)) \ \Rightarrow$$
$$((e \ \overrightarrow{rf} \ d) \ \wedge \ (\neg \ e \ \overrightarrow{rf} \ g)) \ \vee \ ((e \ \overrightarrow{rf} \ g) \ \wedge \ (\neg \ e \ \overrightarrow{rf} \ d)).$$

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

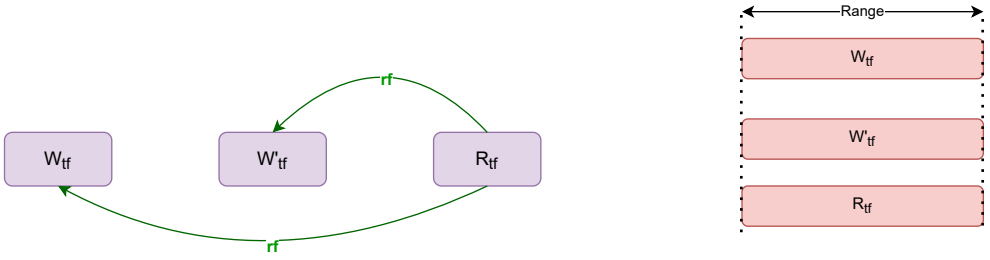


Figure 4.17: Pattern of Tear-free reads

Axiom 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

[9]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'.

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.

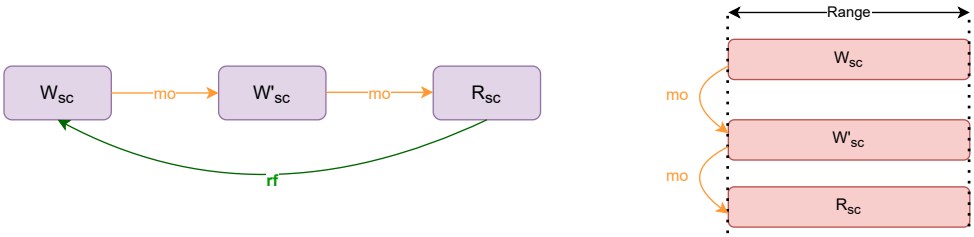


Figure 4.18: A read value cannot come from a write, if there exists a write memory ordered between them and all 3 events are sequentially consistent with equal ranges.

- *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.

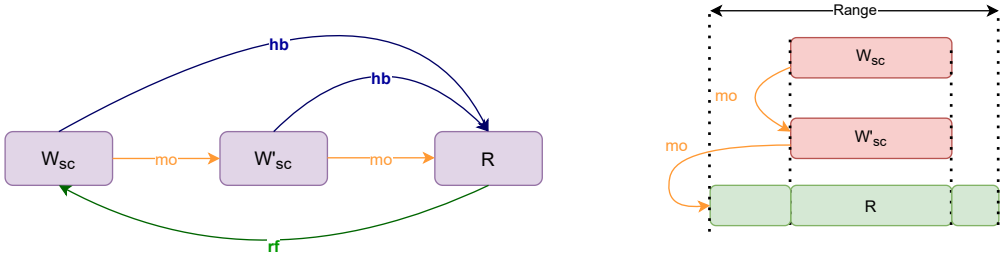


Figure 4.19: A read value cannot come from a write, if there exists a write memory ordered between them and both writes are sequentially consistent with equal ranges.

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

$$g\!:\!sc \ \wedge \ e\!:\!sc \ \wedge \ (\Re(g)\!=\!\Re(e)) \ \wedge \ d \ \overrightarrow{hb} \ g \ \wedge \ d \ \overrightarrow{hb} \ e \ \Rightarrow \ \neg \ e \ \overrightarrow{rf} \ d.$$

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

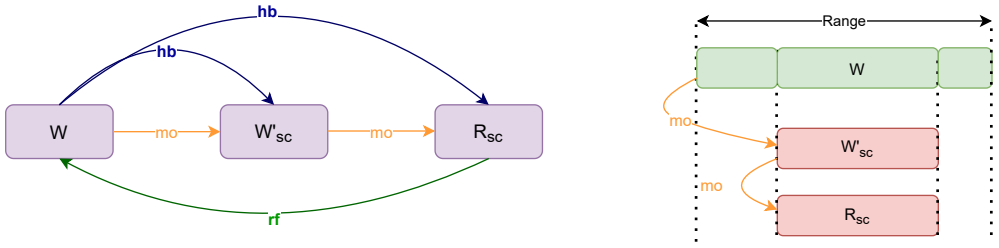


Figure 4.20: A read value cannot come from a write, if there exists a write memory ordered between them and both this write and the read are sequentially consistent with equal ranges.

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 $\overrightarrow{hb}$ relation with each other, and which are either two writes or the two events are involved in a $\overrightarrow{rf}$ relation with each other. This can be stated concisely as,

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

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)

Consistent executions are those which should ideally be possible if the program is actually run on some hardware. For a sequential program, we use the semantics of the programming language to understand what can be the outcome of a program. For a concurrent program, since we can have multiple outcomes of the same program being executed (keeping all inputs constant), we need a semantic model to rely on. The memory model is in essence just this semantic model for programs using shared memory.

In our language, a consistent execution maps to a valid observable behavior, as this is what the user can actually record as an outcome of the program.

As per the standard specification, valid observable behaviour is when[10]:

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

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

As a summary, this chapter axiomatically defined the ECMAScript memory model. The model is defined using binary relations on events and specifying the constraints of the model in terms of restricting reads-from relations given other binary relations

[10]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.

that can exist between events in a Candidate Execution. In the next chapter, we use this formal model to reason about the validity of instruction reordering under the constraints of the model.

Chapter 5

Instruction Reordering

This chapter adddresses the validity of instruction reordering under the ECMAScript memory model. We first start by showing some examples of Candidate Executions where reordering, while sequentially sane to do, it not safe in the relaxed memory context. We then summarize our approach towards a proof to identify when such a reordering is safe with respect to shared memory accesses. Next, we introduce two more defintions for our purpose followed by two basic lemmas that will be instrumental for proofs in this chapter and the next. Next, we formulate a theorem and a corresponding corollary to assess validity of reorderinng at a Candidate Execution level. Lastly, we address conservatively at the program level (still abstracted to a set of shared memory events) involving loops and conditional branching. Throughout this chapter, we use counter examples to give a better intuitive understanding of the elements of the proof as well as the advantage of our formal model in Chapter 3.

5.1 Introduction

Instruction reordering is a common operation done by the compiler / hardware for 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 $\overrightarrow{hb}$ relations.

Some Examples We show a couple of examples to showcase why reordering may not be that straightforward.

Consider the first example below of a Candidate and the resultant candidate after reordering two events:

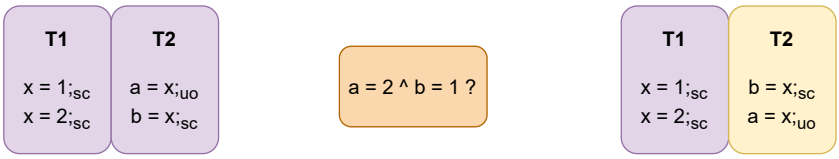


Figure 5.1: Ex1(a)

The figure on the left is the original candidate and that on the right is after reordering the two reads of $T2$. The observable behavior in question is written in the middle.

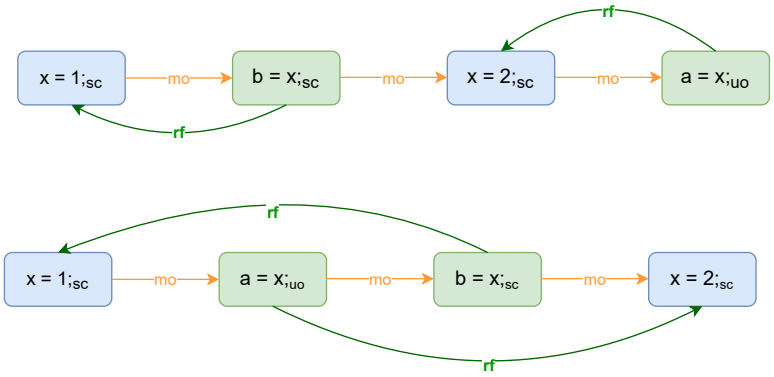


Figure 5.2: Ex1(b)

The above figure has two sets of relations. The first justifies the outcome for the reordered candidate. While the second justifies the first. Notice that for the second, one may have a read memory ordered before a write that it reads from. This is quite

counter intuitive to understand at first. But strictly from the semantics of the model, this justification of the observable behavior is completely valid.

Consider another example below:

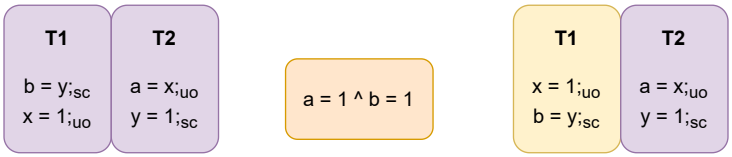


Figure 5.3: Ex2(a)

The figure on the left is the original candidate and that on the right is after reordering the two events of $T1$. The observable behavior in question is written in the middle.

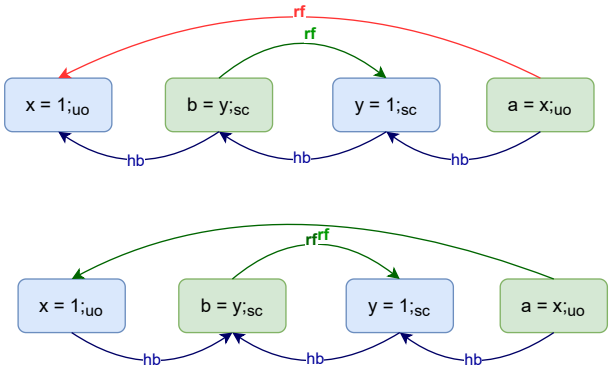


Figure 5.4: Ex2(b)

The above figure has two sets of relations. The first justifies that such an outcome is not possible for the original program candidate due to Axiom 1. While the second justifies that this outcome is possible for the reordered program.

Note that we cannnot infer in the reordered candidate the set of relations for any candidate execution to have $a = x;_{uo} \overrightarrow{hb} x = 1;_{uo}$.

The point of the above two examples is to show that we have to be careful while reordering two events in the same thread. By example case analysis, for each observable behavior, one must check all possible candidate executions and assert whether such an observable is possible or not. This method of checking validity of reordering will scale exponentially as the program size increases. It is often also the

case that the compiler may not have information on which exact events would be executed in other threads to assert such reordering is valid or not.

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

5.2 Summary of Our Approach

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 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 $\overrightarrow{rf}$ (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.

We view reordering as manipulating the agent-order relation. 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.

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.

5.3 Preliminaries

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

Definition 5.3.1. *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*

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

Definition 5.3.2. *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* $dir(e,d) \ \Rightarrow \ cons(e,d) \ \vee \ e \overrightarrow{_{sw}} \ d$
- *If* $d{:}sc \ \wedge \ d{\in}W$*, then* $dir(e,d) \ \Rightarrow \ cons(e,d)$
- *If* $d{:}sc \ \wedge \ e{\in}R$*, then* $dir(e,d) \ \Rightarrow \ cons(e,d) \ \vee \ e \overrightarrow{_{sw}} \ d$

Definition 5.3.3. *Reorderable Pair (Reord)*

We define a boolean function Reord *that takes two ordered pair of events e and d such that* $e \overrightarrow{_{ao}} \ d$ *and gives a boolean value indicating if they are a reorderable pair.*

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

5.4 Useful Lemmas

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

$$cons(e,d) \ \wedge \ e \overrightarrow{_{ao}} \ d \ \wedge \ ((d{:}uo) \ \vee \ (d{:}sc \ \wedge \ d{\in}W))$$

then,

$$k \overrightarrow{_{hb}} \ d \Longrightarrow k \overrightarrow{_{hb}} \ e$$

When we have two consecutive events e and d which are one after the other (i.e. $e \overrightarrow{_{ao}} \ d$*), we can use transitive property of* $\overrightarrow{_{hb}}$ *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 :

$$cons(e,d) \ \wedge \ e \overrightarrow{_{ao}} \ d \tag{0}$$

In the first case, we have $d : uo$. From Def 5.3.2 and Def 5.3.1, we have for any event k

$$dir(k, d) \Rightarrow cons(k, d)$$

From (0), we have $k = e$ satisfying the above property with d. Because $\overrightarrow{ao}$ is a total order, e will be the only event. Thus, for any other $k \neq e$, we have

$$k \overrightarrow{hb} d \Rightarrow k \overrightarrow{hb} d$$

The following figure should explain this intuition:

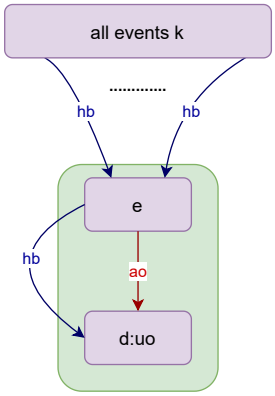


Figure 5.5: For the first case

In the second case, we have

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

Thus, from Def 5.3.2 and Def 5.3.1, for any event k, we have

$$dir(k, d) \Rightarrow cons(k, d)$$

From (4), event e satisfies the above. Though there could be direct *happens-before* relation with some event k from another *agent*, from Def 5.3.2 these are only relations satisfying $dir(d, k)$. Thus, we can once again infer that for any $k \neq e$

$$k \overrightarrow{hb} d \Rightarrow k \overrightarrow{hb} d$$

The following figure explains this intuition:

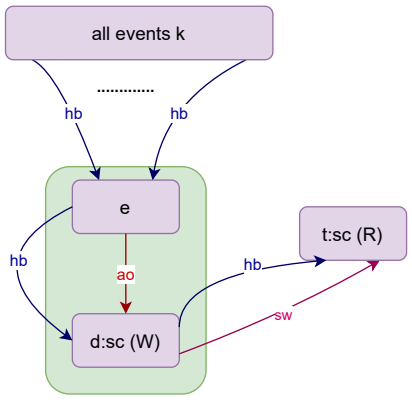


Figure 5.6: For the second case

□

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

If

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

then,

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

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 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, we have $e:uo$. From Def 5.3.2 and Def 5.3.1, we have for any event k

$$dir(e,k) \Rightarrow cons(e,k)$$

From (0), we have $k = d$ satisfying the above property with e. Because $\overrightarrow{ao}$ is a total order, d would be the only such event. Thus, for any other event $k \neq d$, we can infer,

$$e \overrightarrow{hb} k \Rightarrow d \overrightarrow{hb} k$$

The following figure should explain this intuition:

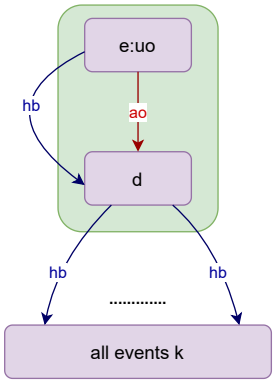


Figure 5.7: Caption

In the second case, we have

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

Thus, from Def 5.3.2 and Def 5.3.1, for any event k, we have

$$dir(e, k) \Rightarrow cons(e, k)$$

From (4), we have event $k = d$ satisfying the above property with e Though there could be direct *happens-before* relation with some event k from another *agent*, from Def 5.3.2 these are only relations satisfying $dir(k, e)$. Thus, we can infer that for any $k \neq d$

$$e \overrightarrow{hb} k \Rightarrow d \overrightarrow{hb} k$$

The following figure explains this intuition:

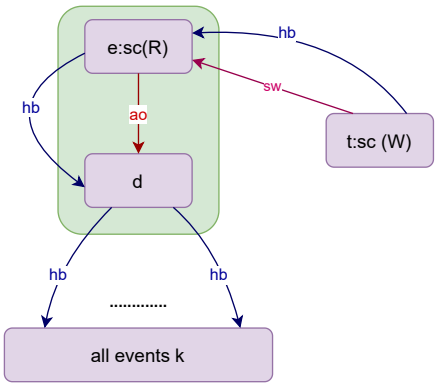


Figure 5.8: Caption

□

5.5 Valid reordering

Our main objective is 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 is to ensure that the Candidate of a program does not have some new behaviours that weren't supposed to happen prior to reordering.

5.5.1 Reordering of Consecutive Events

Theorem 5.1. *Consider a candidate C of a program and its possible Candidate Executions where $\overrightarrow{hb}$ is strictly partial order. Consider two events e and d such that $cons(e, d)$ is true in C and $e \overrightarrow{ao} d$. 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 $\overrightarrow{hb}$ relations (except $e \overrightarrow{hb} d$) and that any new $\overrightarrow{hb}$ 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 $e \overrightarrow{hb} d$ 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 *Candidate Execution* 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.

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

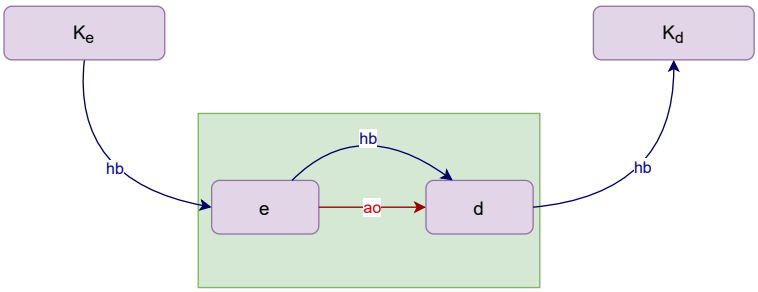


Figure 5.9: For any Candidate Execution of C, the set K_e and K_d

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 dummy events 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 following changes

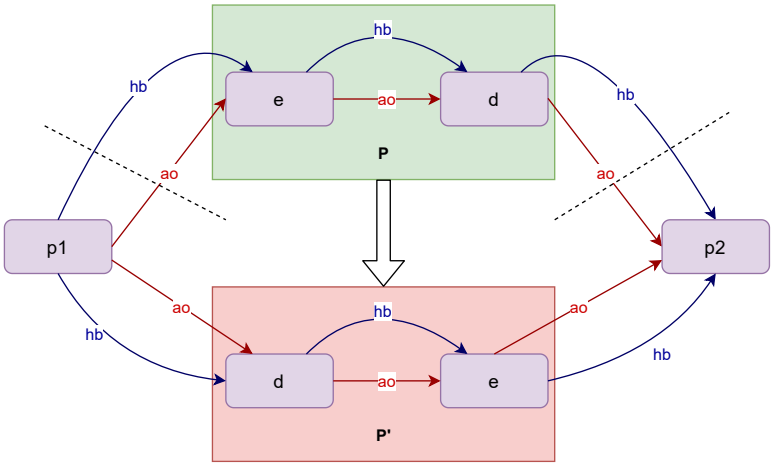


Figure 5.10: The direct relation changes that can be observed while reordering events e and d

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) $p1 \ \overrightarrow{hb} \ e$ b) $e \ \overrightarrow{hb} \ k$

c) $d \ \overrightarrow{hb} \ p2$ d) $k \ \overrightarrow{hb} \ d$

After reordering, we have (a) and (c) preserved due to transitivity

$$p1 \ \overrightarrow{hb} \ d \ \wedge \ d \ \overrightarrow{hb} \ e \ \Rightarrow \ p1 \ \overrightarrow{hb} \ e.$$
$$e \ \overrightarrow{hb} \ p2 \ \wedge \ d \ \overrightarrow{hb} \ e \ \Rightarrow \ d \ \overrightarrow{hb} \ p2.$$
$$p1 \ \overrightarrow{hb} \ d \ \wedge \ d \ \overrightarrow{hb} \ e \ \wedge \ e \ \overrightarrow{hb} \ p2 \ \Rightarrow \ p1 \ \overrightarrow{hb} \ p2.$$

(b) and (d) may not be preserved due to $d \ \overrightarrow{sw} \ k$ or $k \ \overrightarrow{sw} \ d$. 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

$$\forall \ k \in K_e - \{p1\}, \ k \ \overrightarrow{hb} \ p1.$$
$$\forall \ k \in K_d - \{p2\}, \ p2 \ \overrightarrow{hb} \ k.$$

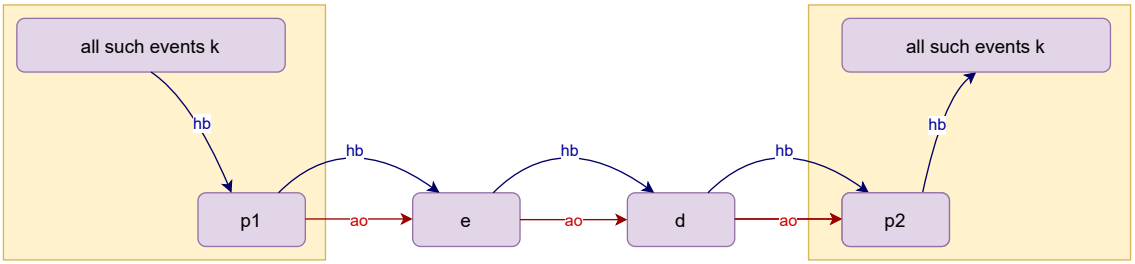


Figure 5.11: For any Candidate execution, the intuition behind valid pivots $< p1, p2 >$

By Lemma 1 and 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[1]$< p1, p2 >$

<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 5.12: 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 that in this example, $K_e = K_{e1} + K_{e2} + p1 + p_x$

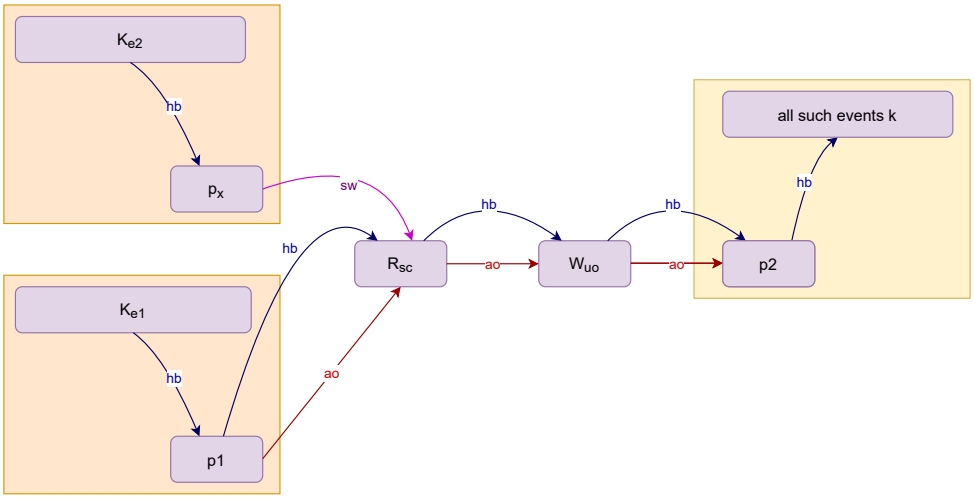


Figure 5.13: A Candidate Execution where p1 is not a valid pivot

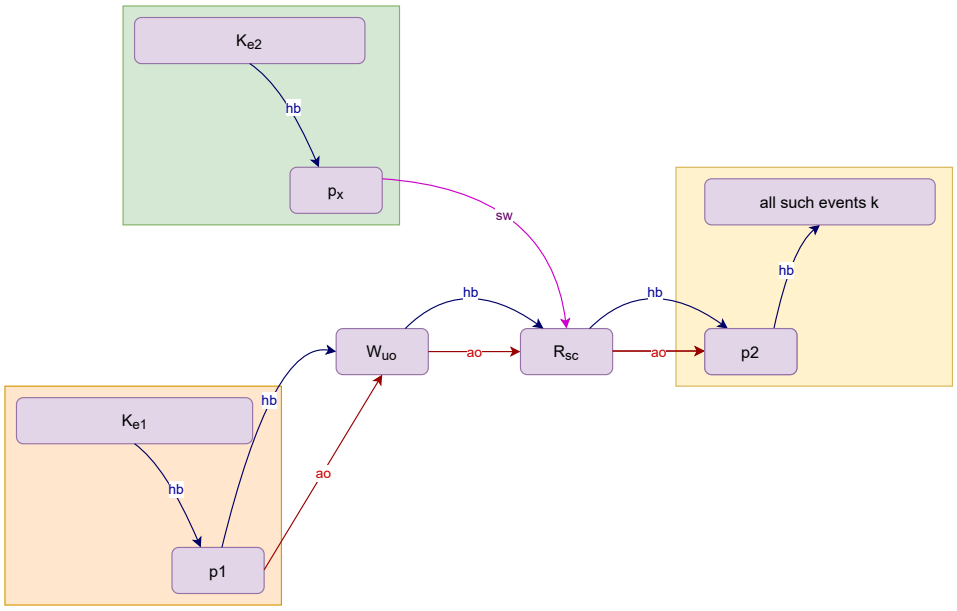


Figure 5.14: The resultant Candidate Execution after reordering, exposing the relations with p_x, K_{e2} and d that arelost

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 *happens-before* relations that come through *synchronize-with* relations wiht d. Thus, although we are able to preserve relations that existed in any $CandidateExecution$ of C, we also in the process, introduce new ones in

[1]This proof does not go about showing the exact happens-before relations that are preserved; rather it uses the properties between different happens before relations that hold, which would imply that for any possible Candidate Execution after reordering, the set of happens-before relations apart from that between e and d remain the same.

CandidateExecutions of C'. The figure below shows pictorially an example of a Candidate Execution of C for the case above

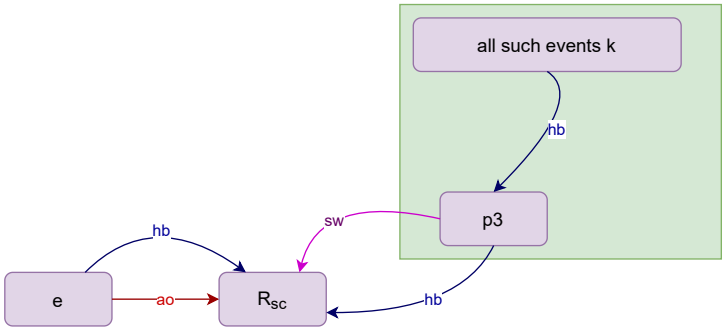


Figure 5.15: A Candidate Execution where d is a sequentially consistent read

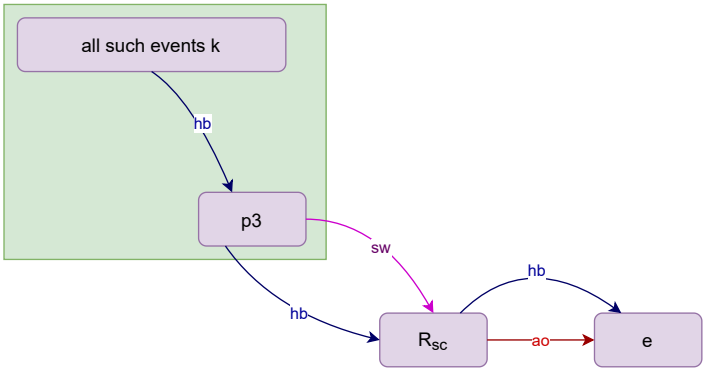


Figure 5.16: The Candidate Execution after reordering, exposing the new relations established with e, $p3$ and set k

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

For these cases, we must know whether the 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 $\overrightarrow{hb}$ cycles introduced in the process. Consider the example below

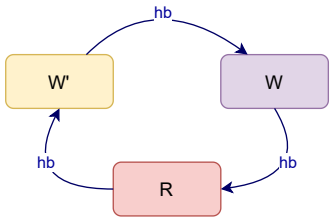


Figure 5.18: Caption

Fix the above figure to put event labels in lowercase.

Notice that here, Axiom 1 restricts read r to read from w'.

$$r \overrightarrow{hb} w' \Rightarrow \neg r \overrightarrow{rf} w'.$$

By transitive property of happens-before, it is also the case that $w' \overrightarrow{hb} r$.

$$w' \overrightarrow{hb} w \ \wedge \ w \overrightarrow{hb} r \ \Rightarrow \ w' \overrightarrow{hb} r.$$

As per this, Axiom 1 will not restrict $r \overrightarrow{rf} w'$. To avoid such cases, we will need to ensure that no Candidate Execution of C' after e and d are reordered have $\overrightarrow{hb}$ 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[2].

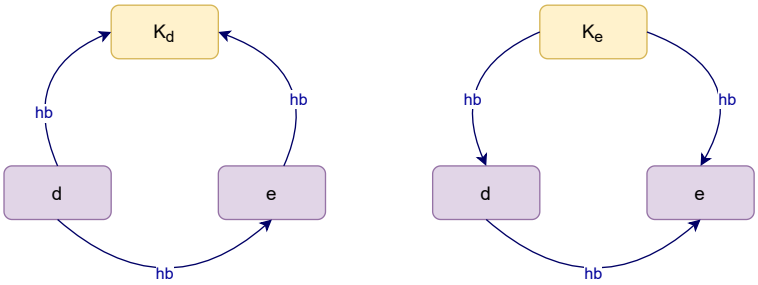


Figure 5.19: If k belongs to one of the sets K_e or K_d

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.

[2] 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 relations they are covered by K_e and K_d.

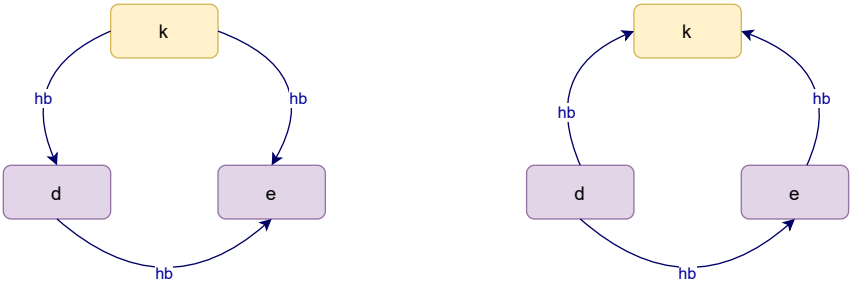


Figure 5.20: If $k \xrightarrow[hb]{} e$ or $d \xrightarrow[hb]{} k$ are new sets of relations

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, $e \xrightarrow[hb]{} k$[3]. Thus, we have a cycle. The following figure shows this pictorially

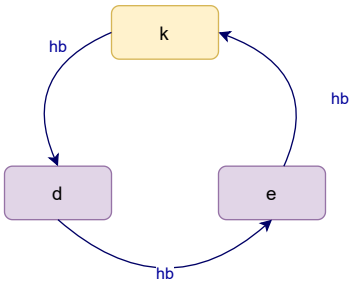


Figure 5.21: A cycle exists in the case where we have two new sets of relations ($k \xrightarrow[hb]{} e$ and $d \xrightarrow[hb]{} k$)

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

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

[3]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.

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 $d \xrightarrow[hb]{} k$, Suppose a cycle exists with another event k'. Then

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

Note that the latter two relations are not new, since the only new set of relations are of the first form. Now, by Lemma 2 and by transitivity respectively, we have

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

Thus, we also have the cycle

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

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.

To summarize, the table below shows the cases where new relations have no happens-before cycles.

New Reln	R-R	R-W	W-R	W-W
uo-uo	NO	NO	NO	NO
uo-sc	YES	NO	YES	NO
sc-uo	NO	NO	YES	YES
sc-sc	NO	NO	YES	NO

Table 5.1: Insert good caption here.

Fix the table to replace "YES" with "Y" and "NO" with "N".

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.

Let us first consider the variants of events e and d that we need to analyze from the previous table:

a) $e \in R \ \wedge \ d \in R \ \wedge e : uo \ \wedge d : uo$

b) $e \in R \ \wedge \ d \in R \ \wedge e : uo \ \wedge d : sc$

c) $e \in R \ \wedge \ d \in W \ \wedge e : uo \ \wedge d : uo$

d) $e \in W \ \wedge \ d \in R \ \wedge e : uo \ \wedge d : uo$

e) $e \in W \ \wedge \ d \in R \ \wedge e : uo \ \wedge d : sc$

f) $e \in W \ \wedge \ d \in R \ \wedge e : sc \ \wedge d : uo$

g) $e \in W \ \wedge \ d \in W \ \wedge e : uo \ \wedge d : uo$

h) $e \in W \ \wedge \ d \in W \ \wedge e : sc \ \wedge d : uo$

We analyze each of the above case one by one by first considering the original relation ($e \xrightarrow[hb]{} d$) and the reordered one ($d \xrightarrow[hb]{} e$).

- (a) and (b) do not fit any pattern of our Axioms, hence even after reordering the agent order between them does not match any other axiom. Hence this relation does not introduce any new observable behavior. This is irrespective of the range between the two read events.

- (c) fits in the pattern of Axiom 1, when they have at least overlapping ranges. Before reordering, d is not allowed to read from e, but after reordering, it can. Hence this relation can introduce observable behavior if the range between events e and d at least overlap.

- (d), (e) and (f) can fit in the pattern of Axioms 1 and 3, if e and d at least have overlapping ranges, preventing d from reading parts of e or some parts of another write k due to e being the intervening write. But after reordering, d is allowed to read parts of k, which introduces new observable beahviors.

- (g) and (h) can fit in the pattern of Axioms 1 and 3, if they have at least overlapping ranges. Before reordering, the agent order between e and d could prevent some read k from reading parts of e. This is not the case after reordering, thus possibly introducing a new observable behavior.

In summary, on observing the role on the Axioms on the relation between e and d, 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 at least overlapping ranges, one could introduce a new observable behavior after reordering them.

We will later show counter examples for each of the above cases that we discard as invalid to reorder. Decide whether to put figures here explaining each pattern that matches or palce counterexamples. The latter will be too much.

Any other new relations that are introduced can be divided into 4 cases, in terms of our events e and d and the new relation 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.

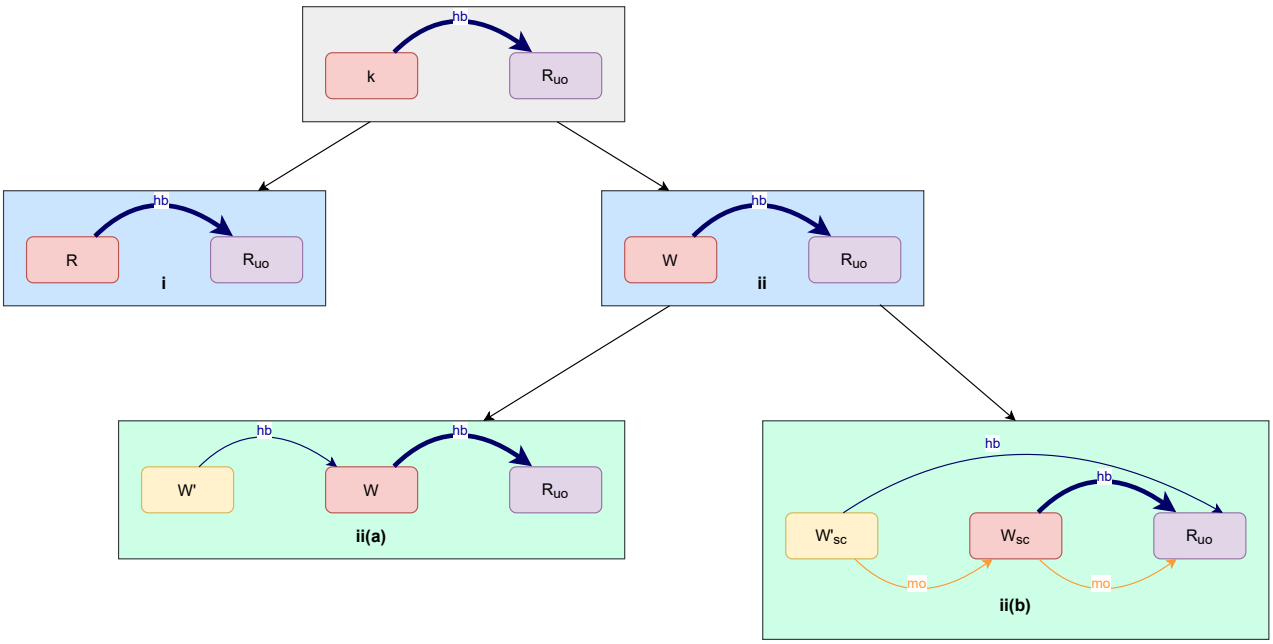


Figure 5.22: The role of the axioms on introducing a new relation between an unordered Read and some event k

- For (i), when k is a read, the pattern matches none of the Axioms.
- For (ii), when k is a write, Axiom 1 (ii(a)) or Axiom 3 (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 case (b), varying based on the nature of event k.

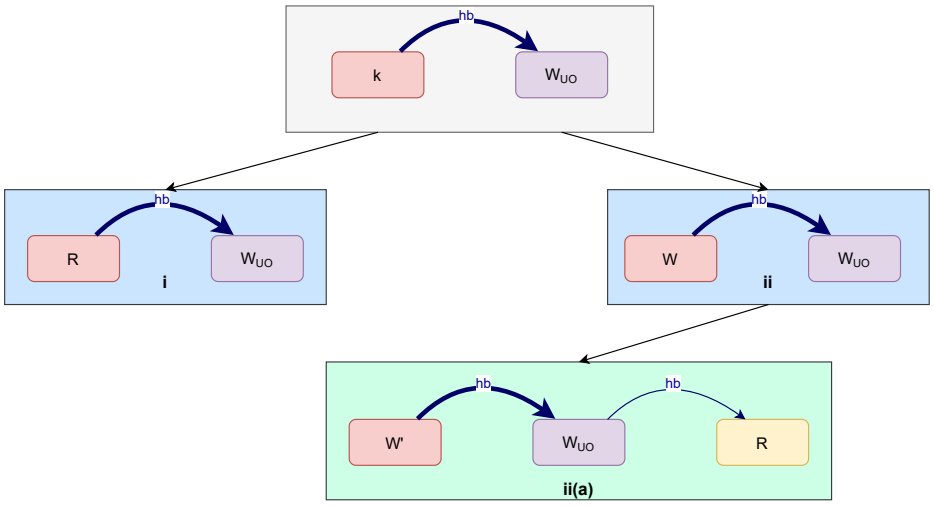


Figure 5.23: (i) and (ii(b)) satisfy the axiom of Coherent Reads

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

- For (i), when k is a read, Axiom 1 restricts k from reading from the write e.
- For (ii), when k is a write, Axiom 1 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.

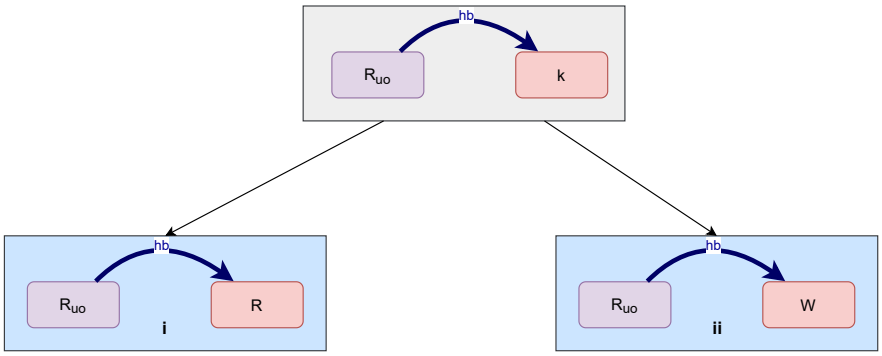


Figure 5.24: (ii) satisfies the axiom of Coherent Reads

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

- Case (i) does not correspond to any pattern restricted on the model, thus having no impact on the observable behaviors.
- For (ii), when k is a write, Axiom 1 restricts the read d from reading values of write k.

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

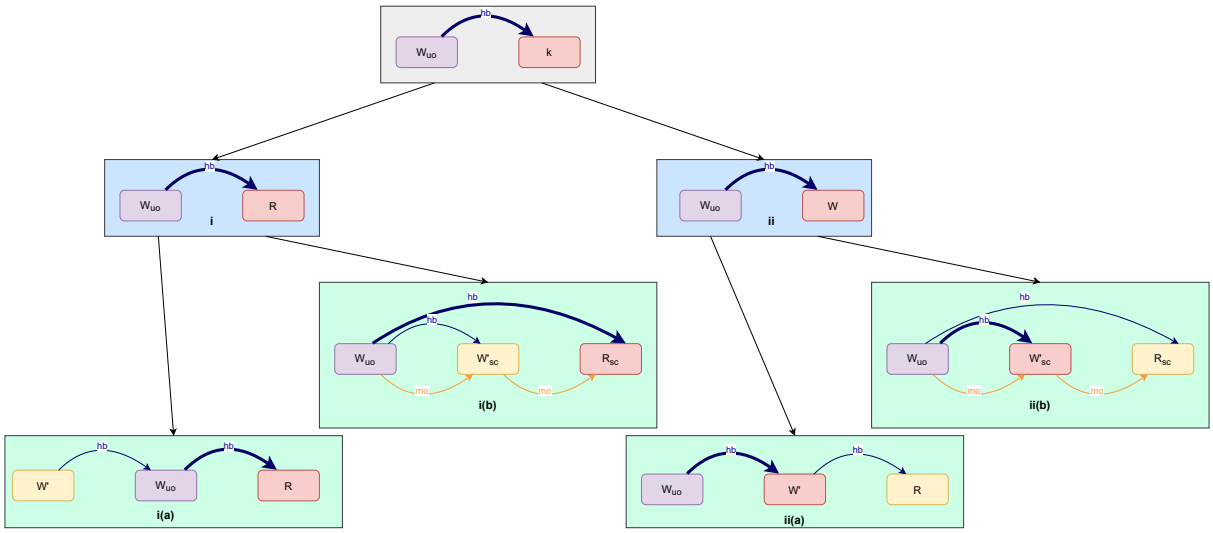


Figure 5.25: (i(a)), (ii(a)) satisfy the axiom of Coherent Reads, whereas (i(b)), (ii(b)) satisfy the axiom of SequentiallyConsistent Atomics

For case (d) we can observe the following

- For case (i), Axiom 1 (i(a)) or Axiom 3 (i(b)) could restrict a read k from reading values of write d,
- For case (ii), Axiom 1 (ii(a)) or Axiom 3 (ii(b)) could restrict a read from reading values of write d,

The above case wise analysis showed us that any new relation (apart from $d \xrightarrow{hb} e$), matching the patterns of the axioms, only enables in restricting possible observable behaviors, which are $\xrightarrow{rf}$ relations. Thus, we can infer that no new observable behavior is introduced due to the new set of $\xrightarrow{hb}$ relations.

In summary, 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 5.26: The final table summarizing the valid cases where observable behaviors will only be a subset after reordering.

Need to label all figures and refer them properly. Also consider elaborating a bit more on each subcase.

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.

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

□ □

5.5.2 Reordering Non-Consecutive Events

Now that we know when two consecutive events can be reordered, we shift our focus to the general reordering of events in an agent. The following corollaries cover all those cases.

Corollary 5.1.1. *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* $e \overrightarrow{_{ao}} d$*. Consider another Candidate C' resulting after reordering e and d in C. If*

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

then, the set of Observable behaviors of C' is a subset of C.

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 our candidate C is

$$e \xrightarrow[ao]{} k \xrightarrow[ao]{} d \wedge \ cons(e, k) \ \wedge \ cons(k, d).$$

What we want after reordering is C', with

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

whose observable behaviors is subset of C.

Without loss of generality, we can choose to first reorder k and d. With $Reord(k, d)$, we can, from Theorem 5.1, reorder them, giving us candidate C'' with

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

whose observable behaviors is subset of C.

Similarly, now with $Reord(e, d)$, by Theorem 5.1, we get candidate C''' with

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

whose observable behaviors is subset of C''.

Now lastly, we need to reorder e and k for which we need $Reord(e, k)$ to hold, thus by Theorem 5.1, giving us our final candidate C' with

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

whose observable behaviors is subset of C'''.

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

2. Inductive Case $n > 1$ Assume the above corollary holds for $n = t$, meaning the observable behaviors of candidate C'_t is a subset of C_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 by Theorem 5.1, giving us the resultant candidate C_t with

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

whose observable behaviors is a subset of C_{t+1}.

Now we have t such events between e and d. With our assumption, we can reorder e and d, thus giving us candidate C'_t with

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

whose observable behaviors is a subset of C_t.

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 by Thoerem 5.1, giving us finally candidate C'_{t+1} with

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

Whose observable behaviors are a subset of C'_t.

By transitive property of subsets, we can conclude that the Observable Behavior of the final candidate C'_{t+1} after reordering is a subset of C_{t+1}.

Hence, by induction the proof is complete. □

Corollary 5.1.2. *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 $k_i \xrightarrow[ao]{} k_{i+1} \wedge cons(k_i, k_{i+1})$. Consider an event e such that*

$$cons(e, k_1) \ \wedge \ e \xrightarrow[ao]{} k_1.$$

Consider another candidate C' with the only differnence from C being $cons(e, k_n) \ \wedge k_n \xrightarrow[ao]{} e$. If

$$\forall i \in [1, n] \ . \ Reord(e, k_i).$$

then the set of observable behaviors of C' is a subset of that of C

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

Corollary 5.1.3. *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 $k_i \xrightarrow[ao]{} k_{i+1} \wedge cons(k_i, k_{i+1})$. Consider an event d such that*

$$cons(d, k_n) \ \wedge \ k_n \xrightarrow[ao]{} d$$

Consider another candidate C' with the only differnence from C being $cons(d, k_1) \ \wedge d \xrightarrow[ao]{} k_1$. If

$$\forall i \in [1, n] \ . \ Reord(k_i, d)$$

then the set of observable behaviors of C' is a subset of that of C

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

Consider whether to write the proof for code motion or not as it is straightforward induction.

5.5.3 Counter Examples for all 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.

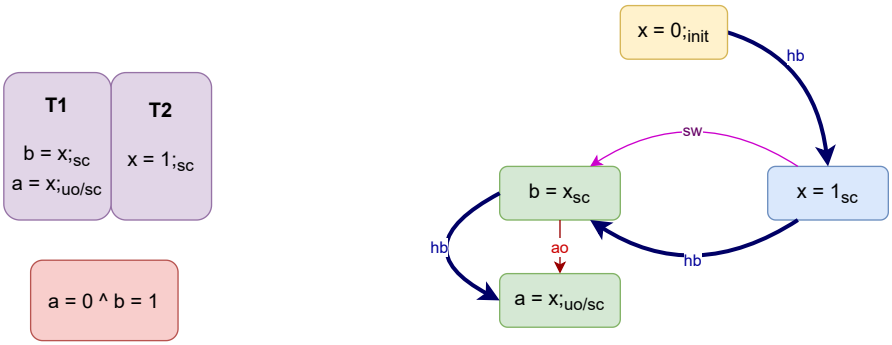


Figure 5.27: Case where a = 0 , b = 1 is invalid due to Coherent Reads

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 $\{x = 0_{init}\} \xrightarrow[hb]{} \{x = 1_{sc}\} \xrightarrow[hb]{} \{a = x_{uo/sc}\}$.
- By the Axiom 1, 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.

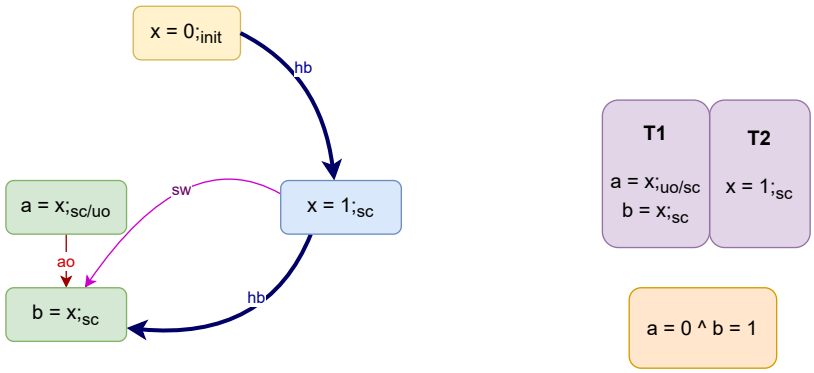


Figure 5.28: Case where the reads are reordered and a = 0 , b = 1 is valid

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.

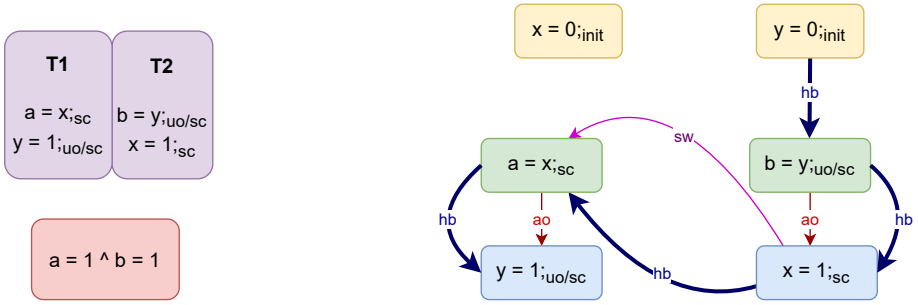


Figure 5.29: Case where a = 1 and b = 1 is invalid due to Coherent Reads.

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 Axiom 1, b cannot read the value of 1 as y.
- This inference was due to $x = 1_{sc} \xrightarrow{hb} a = xsc$

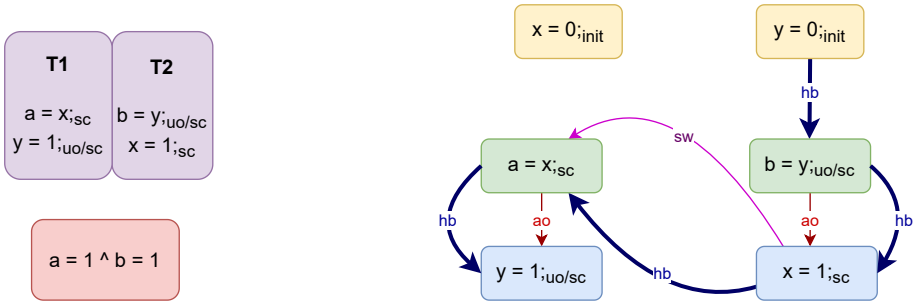


Figure 5.30: Case where events of T1 are reordered, resulting in a = 1 and b = 1 to be valid.

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.

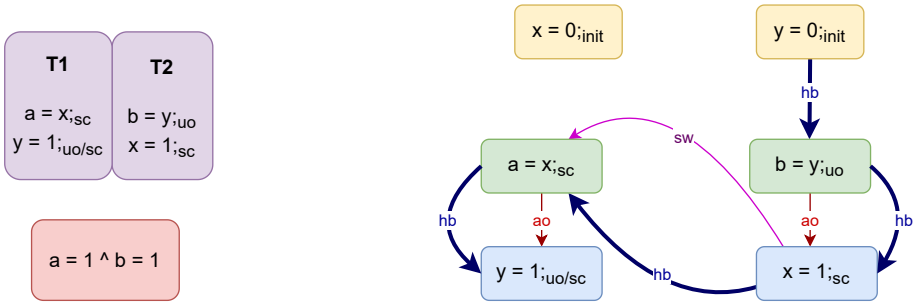


Figure 5.31: Case where a = 1 and b = 1 is invalid due to Coherent Reads.

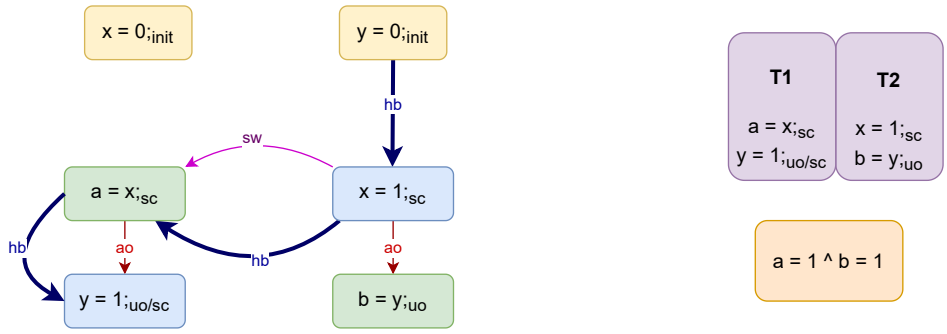


Figure 5.32: Case where events of T2 are reordered, resulting in a = 1 and b = 1 to be valid.

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:

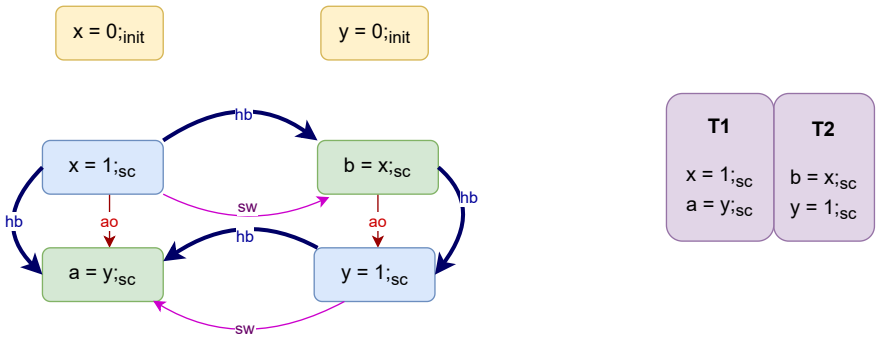


Figure 5.33: Case where a = 1 and b = 1 is valid and no happens-before cycles

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.

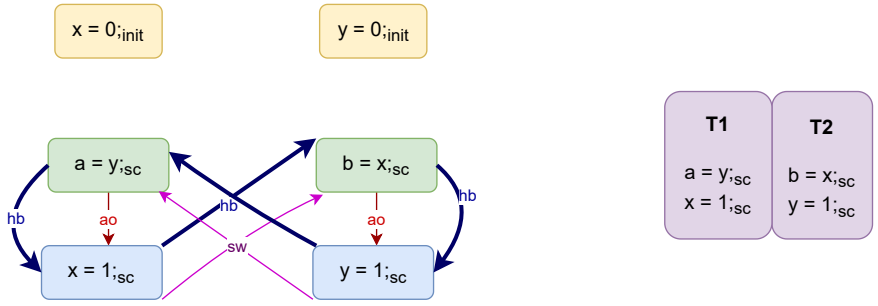


Figure 5.34: Case where a = 1 and b = 1 is creates a happens-before cycle

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.

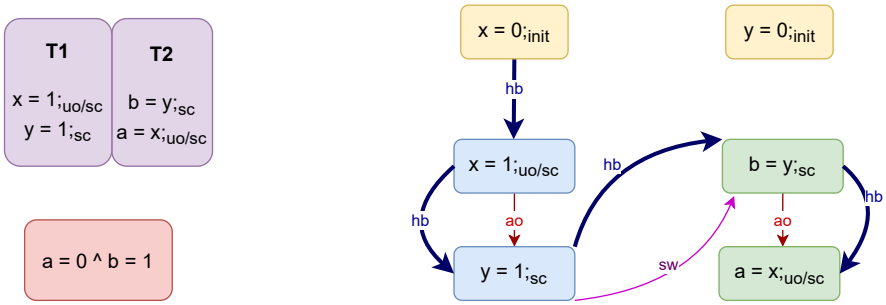


Figure 5.35: Case where a = 0 and b = 1 is invalid due to Coherent Reads.

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 $x = 0_{init} \xrightarrow{hb} x = 1_{uo/sc} \xrightarrow{hb} a = x_{uo/sc}$
- By Axiom 1, the read of a cannot have the value of x read as 0.
- This inference was due to $y = 1_{sc} \xrightarrow{hb} b = y_{sc}$.

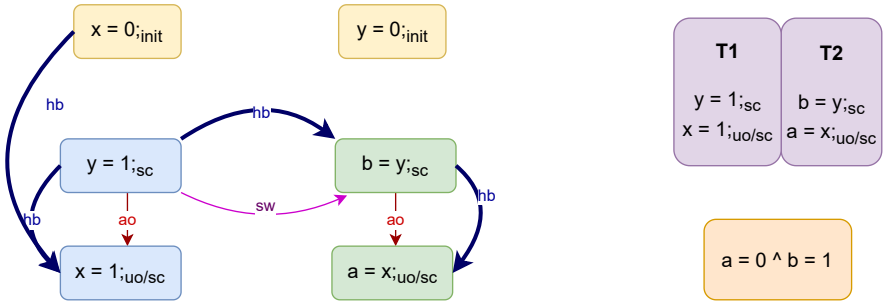


Figure 5.36: Case where events of T1 are reordered, resulting in a = 0 and b = 1 to be valid.

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 $\neg x = 1_{uo/sc} \xrightarrow[hb]{} a = x_{uo/sc}$
- There is no pattern that the Axioms restrict, thus validating x to be read as 0 by a.

All the above shown counter examples only rely on the axiom of coherent reads to show that it is not safe to do the reordering. Does it mean that whenever SC-Atomics axiom can be triggered, Coherent Reads also can be triggered? Investigate this.

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.

5.6 From Candidates to Program

So far we have only addressed reordering at the Candidate level. In practice, a program can have many Candidates. This is due to the program having several conditional branches and loops. To analyze when we can reordering two events at the program level, we must also address the involvement of conditionals and loops that may be between these two events.

The way we approach this is to not have any assumptions as to why the compiler chooses to do a particular reordering in the program. We instead only check if the reoredered program can have its observable behaviors as a subset of the original. This ensures that the algorithm for the compiler optimization need not change, but that our set of conditions will just be additional checks that can be done before actually doing the reordering. Such an approach makes reordering parametric to the memory model.

The downside is that this approach will be conservative as we use no information as to why a particular set of events are reordered. We do not compare and contrast in details the perks of both approaches. This is beyond the scope of this thesis.

5.6.1 Addressing programs with Conditionals

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 Let $B1$ be the sets of events based on a branch of a conditional in a program P. Let C be any Candidate of*

P, Consider $b1$ to be representative of any event in $B1$ and an event k outside the conditional branch. 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[4].

The above property is general for conditionals, being 1-branch or 2-branch. The latter however, has another property which we define below:

Property 2. *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.

The figure below summarizes the two forms of conditionals we can have in any program.

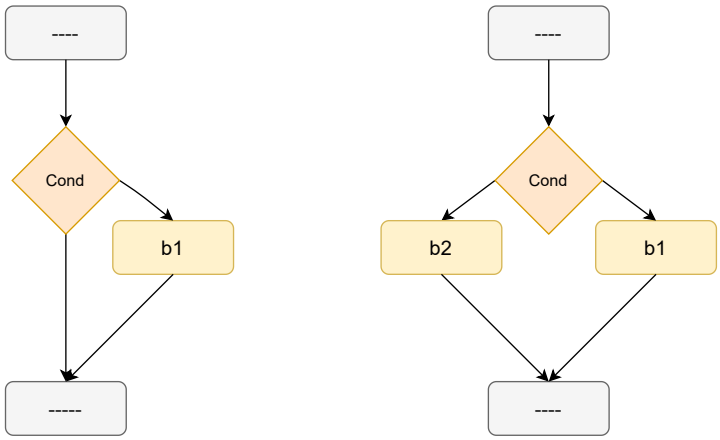


Figure 5.37: Two forms of conditonals

[4]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.

Corollary 5.1.4. *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'. If the following two conditions hold:*

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

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

Proof. 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 2, 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, and neither one of them belong in any conditional branch nested within, then our assumption does not hold.

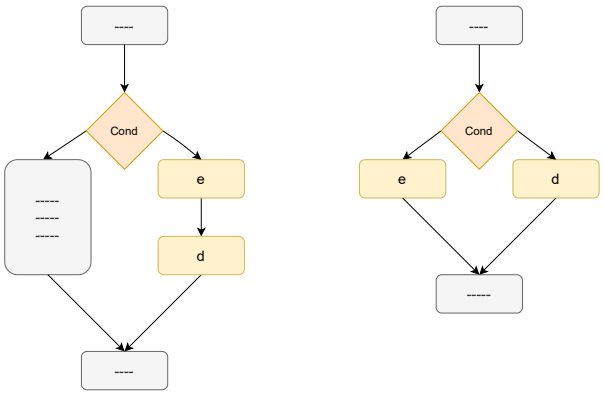


Figure 5.38: The above two cases where our assumption does not hold.

If e and d belong to branches of differnt conditionals, then they can be of two forms, viz. where one conditional branch is nested within the other or both conditonal branches are not nested within each other.

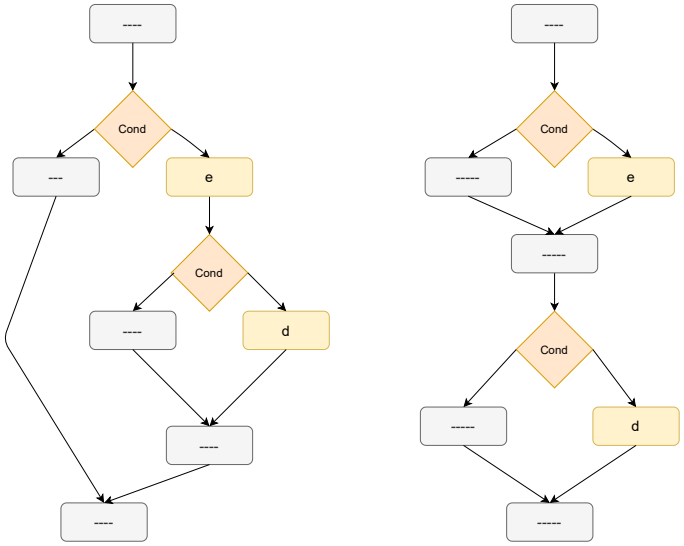


Figure 5.39: Two cases where e and d can both be part of some conditional.

For the first form, without loss of generality, let us consider d is part of conditional branch nested within e's branch.

By Prop 2, there would exist some event l in another branch such that

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

On reodering e and d, we have

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

Thus we have

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

giving us a new Candidate in P' not in P. Irrespective of d 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 1, we would have

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

Suppose there exists another event l in the same branch as d, but not part of any nested conditional within. Then we also have

$$\nexists C \in P \text{ s.t. } (d \in C \ \wedge \ l \notin C) \ \vee \ (d \notin C \ \wedge \ l \in C)$$

The above conclusion need not be proved. Neither should it be put as a property.

On reodering e and d, we have by Prop 1

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

We also have that

$$\exists C \in P' \text{ s.t } (d \notin C \ \wedge \ l \in C)$$

giving us a new Candidate in P' not in P. Irrespective of d 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.

For the second case, suppose they are part of conditonals of typr Prop 2. 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 2, a new observable behavior can be introduced due to a new Candidate that violates the original Prop 2 of every candidate of program P.

Lastly, suppose both e and d are part of conditonal branches satisfying Prop 1, 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 1, 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[5].

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

  By Prop 2, there would exist some event l in another branch such that

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

  On reodering e and d, we have

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

  Thus we have

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

  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.

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

By Prop 1, we would have

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

On reodering e and d, we have

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

Thus we have

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

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

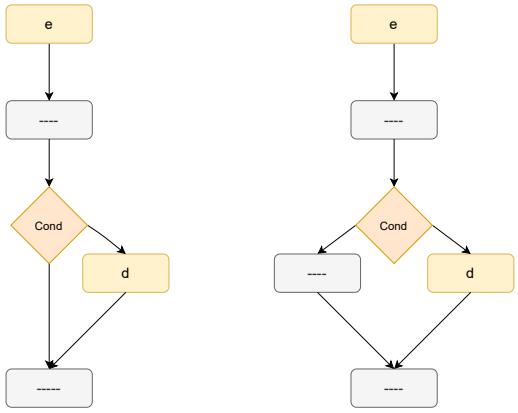


Figure 5.40: Two cases where only d is a part of some conditional branch.

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

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

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.

□

In addition to the above proof, we also show certain counter examples where reordering may not be safe to do. This faciliates better understanding of the proof and its arguments. These counter examples are with respect to reordering of writes.

While showing reordering of reads, one must note that the introduction of new observable behaviors is dependant on the fact that a candidate execution has a local variable which must have not been there beacause the conditional branch was not taken. Note that this fact does not rely on the consistency rules of the memory model. It could be the case that the compiler instantiates the local variables to some default value (say 0), and then decides to reorder a read outside a conditional on the assertion that the read of local variable will return the same constant value. Having such an assertion in general might not be always certain. (Discuss with Clark)

Case when e is not in any conditional branch but d is. One example with two branches. The example below shows a case with a conditonal having two branches where reordering e and d is not safe.

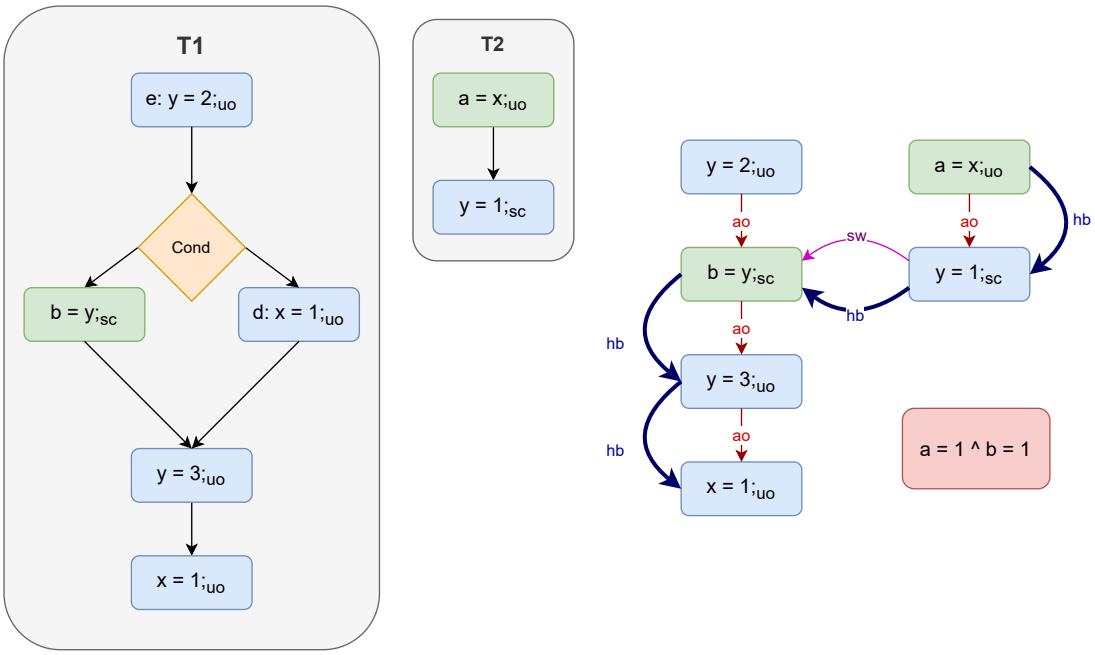


Figure 5.41:

The figure on the left above shows an example of such a program. The figure on the right shows the Candidate Execution where the red box outcome is not allowed when the left branch of the conditional is taken.

- From the Candidate Execution, we can infer $a = x;_{uo} \overrightarrow{hb} y = 1;_{sc} \overrightarrow{hb} b = y;_{sc} \overrightarrow{hb} y = 3;_{uo} \overrightarrow{hb} x = 1;_{uo}$.
- By Axiom 1, the read a cannot have value of x read as 1.
- This inference was due to $a = x;_{uo} \overrightarrow{hb} x = 1;_{uo}$.

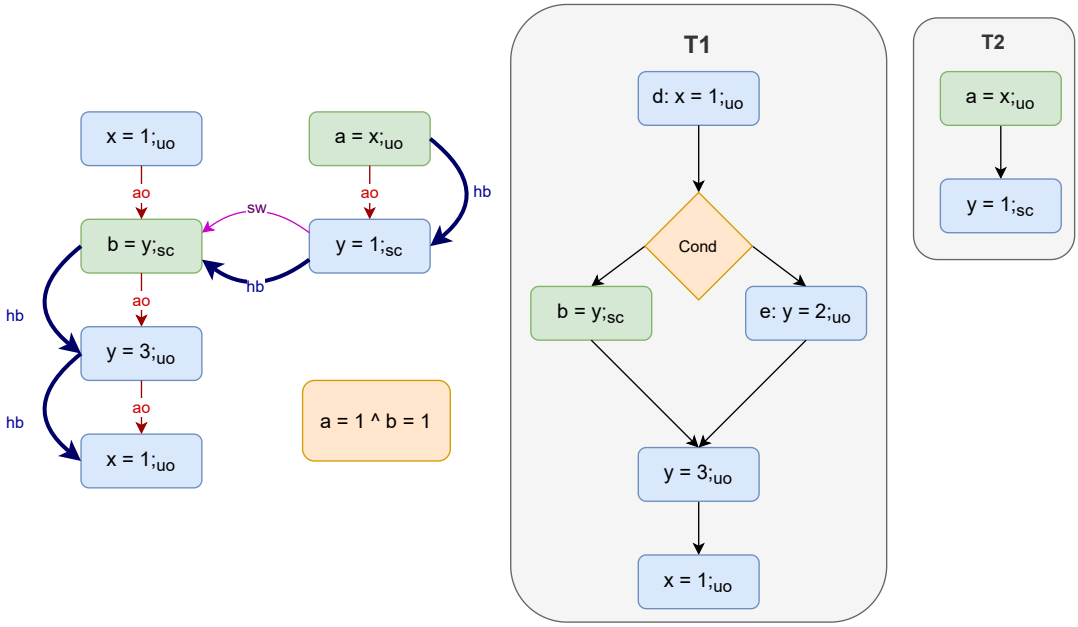


Figure 5.42:

The figure on the right above shows the program after reordering e and d. The figure on the left shows a Candidate Execution where the yellow box outcome is allowed when the left branch of the conditional is taken.

- From the Candidate Execution, we can infer $a = x;_{uo} \overrightarrow{hb} x = 1;_{uo}$.
- But there is no $\overrightarrow{hb}$ relation with event d and the read to x, i.e. $\neg a = x;_{uo} \overrightarrow{hb} d : x = 1;_{uo}$
- No Axiom restricts the read a to have value of x as 1.

One example with one branch.

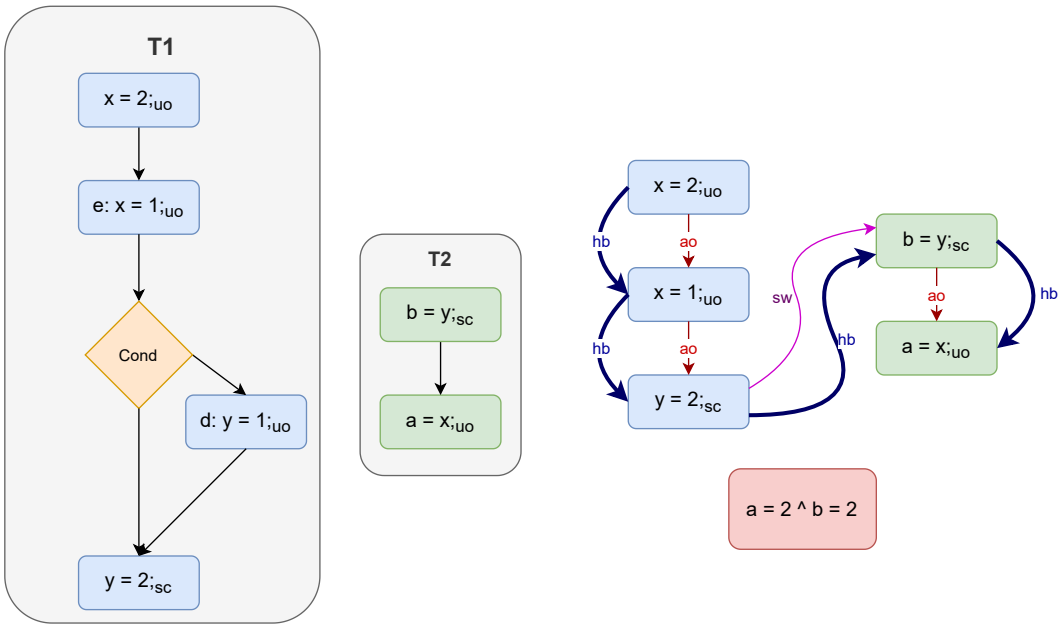


Figure 5.43:

The figure on the left above shows an example of such a program. The figure on the right shows the Candidate Execution where the red box outcome is not allowed when the conditional branch is not taken.

- From the Candidate Execution, we can infer $x = 2;_{uo} \overrightarrow{hb} x = 1;_{uo} \overrightarrow{hb} y = 2;_{sc} \overrightarrow{hb} b = y;_{sc} \overrightarrow{hb} a = x;_{uo}$.
- By Axiom 1, the read a cannot have the value of x to be read as 2.
- This inference was due to the realtion $x = 2;_{uo} \overrightarrow{hb} x = 1;_{uo} \overrightarrow{hb} a = x;_{uo}$.

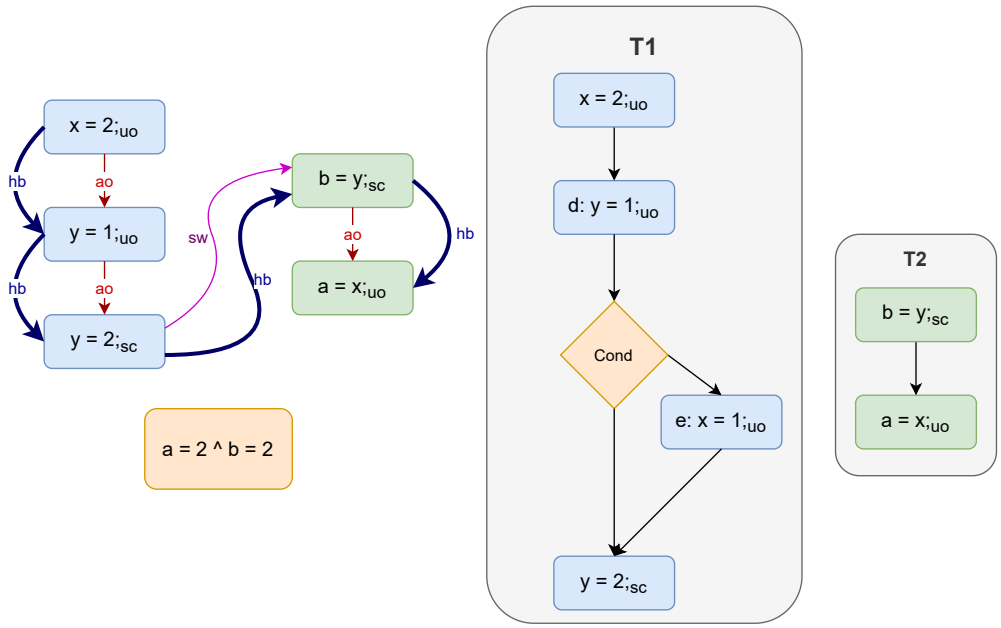


Figure 5.44:

The figure on the right above shows the program after reordering e and d. The figure on the left shows a Candidate Execution where the yellow box outcome is allowed when the conditional branch is not taken.

- From the Candidate Execution, we can infer $x = 1;_{uo} \overrightarrow{hb} a = x;_{uo}$.
- But there is no $\overrightarrow{hb}$ relation with event e and the read to x, i.e. $\neg e : x = 1;_{uo} \overrightarrow{hb} a = x;_{uo}$.
- No Axiom restricts the read a to have value of x as 2.

Case when e is part of a conditonal branch B_e and d is part of another conditonal branch B_d nested within B_e. This case is symmetric to the above. Hence, we do not show another counter example for it. (just consider the code of T1 above to be as a whole under a conditional.)

Case when e and d are part of different conditional branches. One not nested in the other. One example suffices, irrespective of 2 branch or one branch.

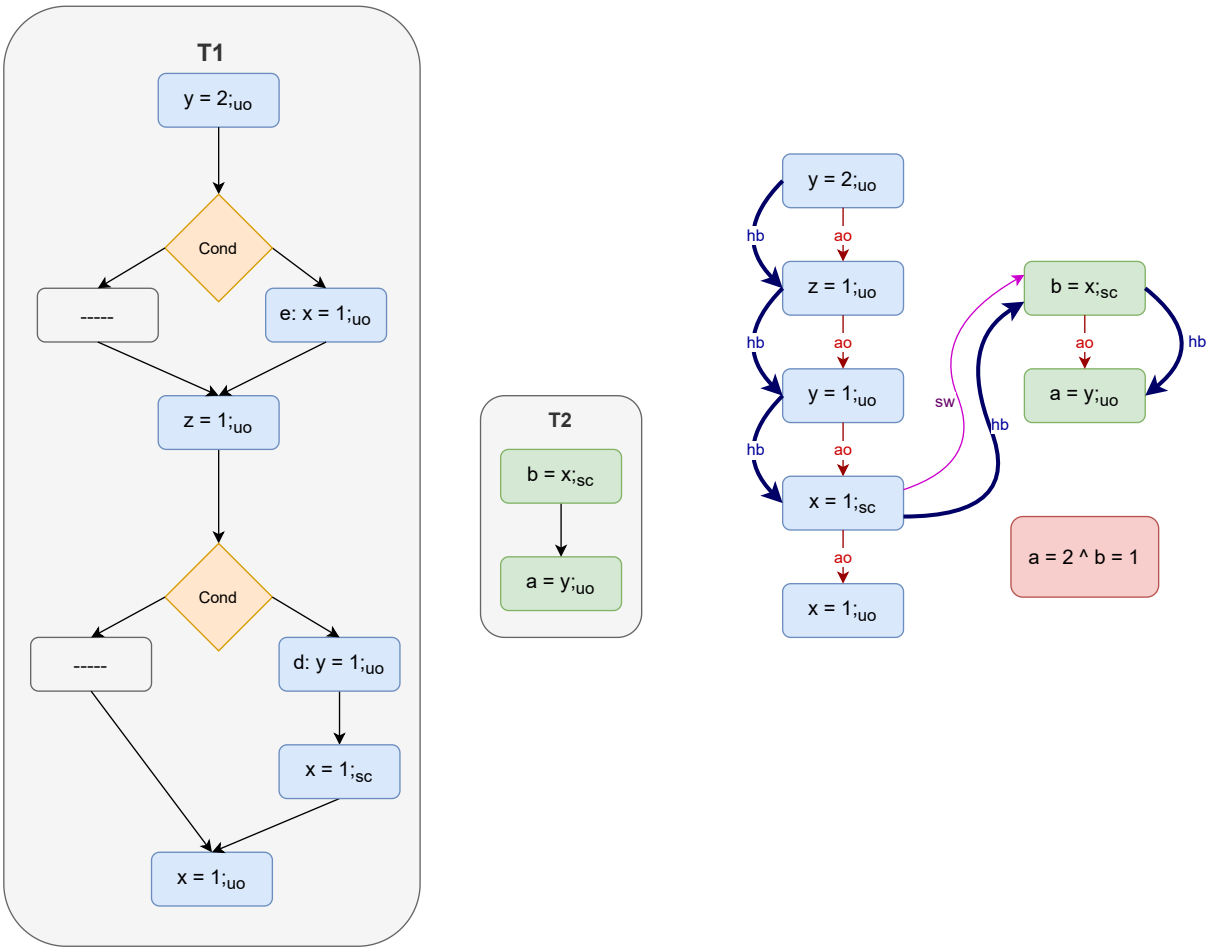


Figure 5.45:

The figure on the left above shows an example of such a program. The figure on the right shows the Candidate Execution where the red box outcome is not allowed when the left branch of the first conditional is not taken while the left branch of the second conditional is.

- From the Candidate Execution, we can infer $y = 2;_{uo} \overrightarrow{hb}\ z = 1;_{uo} \overrightarrow{hb}\ y = 1;_{uo} \overrightarrow{hb}\ x = 1;_{sc} \overrightarrow{hb}\ b = x;_{sc} \overrightarrow{hb}\ a = y;_{uo}$.
- By Axiom 1, the read a to y cannot have the value 2.
- The above inference is due to the relations $y = 2;_{uo} \overrightarrow{hb}\ y = 1;_{uo} \overrightarrow{hb}\ a = y;_{uo}$.

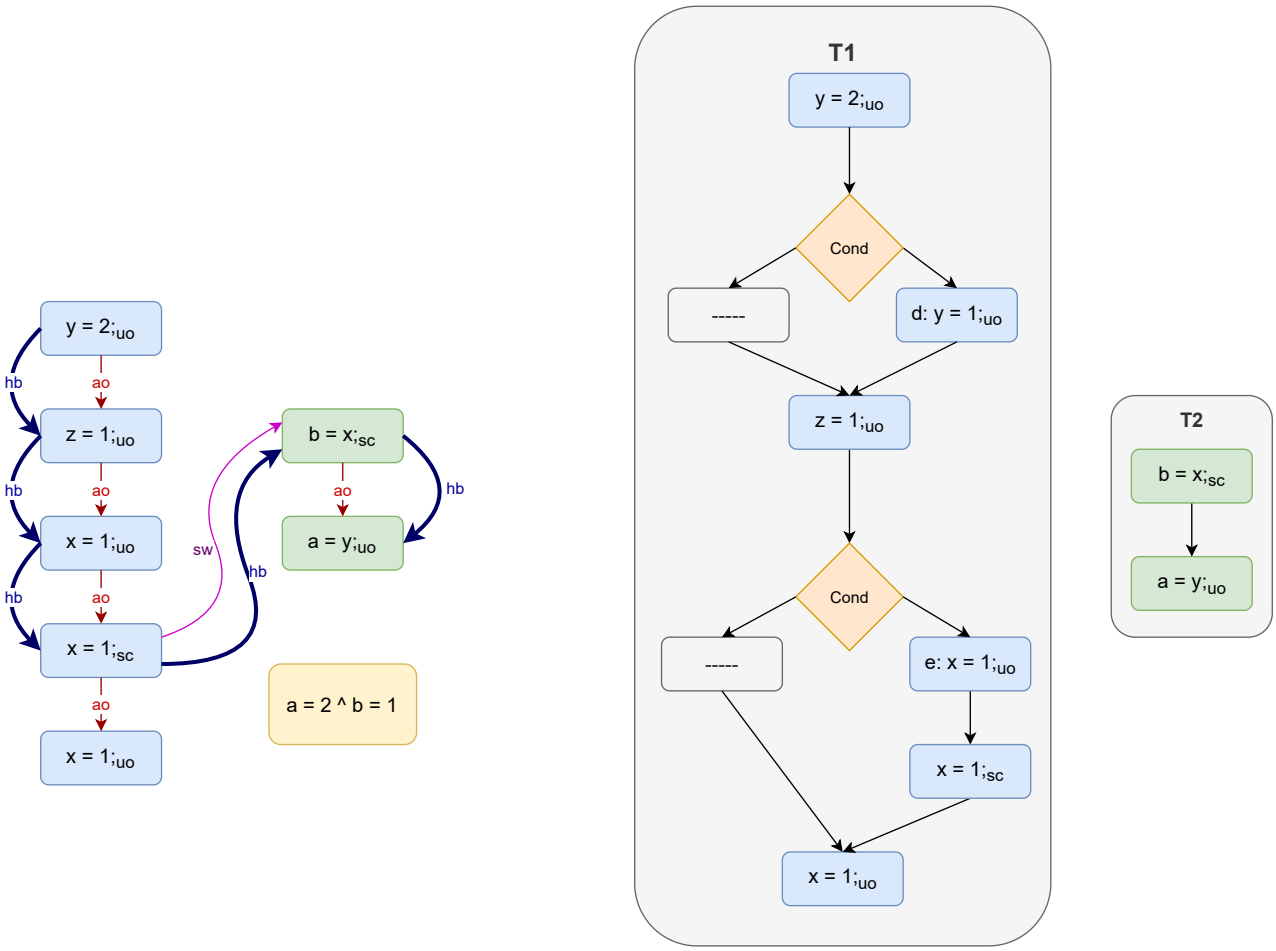


Figure 5.46:

The figure on the right above shows the program after reordering e and d. The figure on the left shows a Candidate Execution where the yellow box outcome is allowed when the left branch of the first conditional is not taken but that of second conditional is.

- From the Candidate Execution, we can infer $y = 2;_{uo} \overrightarrow{hb} a = y;_{uo}$.
- But there is no $\overrightarrow{hb}$ relation with event d and the read to y, i.e. $\neg e : y = 1;_{uo} \overrightarrow{hb} a = y;_{uo}$ and $\neg y = 2;_{uo} \overrightarrow{hb} y = 1;_{uo}$.
- From the above we can infer that no Axiom restricts the read a to have value of y as 2.

5.6.2 Addressing Programs with Loops

Addressing reordering of events in programs with loops is relatively straightforward, leaving one special case. For simplicity (and also Without loss of generality), let us consider our program has only one loop.

There will be one Candidate for each iteration of loop. For convenience, let us define C^i to be a candidate of program with i iterations of the same loop. Let us also define e^i_l to be an event within the loop of the program which in a candidate signifies the i^{th} iteration of the event.

Using the above notation, we have the following property for any consecutive events e and d belonging in a loop:

Property 3.

$$\forall i \neq j,\ Reord(e^i_l, d^i_l)\ \Rightarrow\ Reord(e^j_l, d^j_l)$$

Using the above notation, we have the following corollary for reordering events e and d within a loop

Corollary 5.1.5. *Consider a program P with a loop and its candidates $C^1, C^2, ..., C^n$ in which events e and d are parts of the loop and present in all of them with $e \overrightarrow{_{ao}}\ d$. Consider the set of corresponding candidates $C'^1, C'^2, ..., C'^n$ after reordering e and d in P and its corresponding program P'. If the following three conditions hold:*

$$Reord(e, d)$$
$$\forall C^i \in P,\ \forall j \in [1, i], \forall k\ s.t.\ e^j \overrightarrow{_{ao}}\ k \wedge k \overrightarrow{_{ao}}\ d^j\ .\ Reord(e^j, k) \wedge Reord(k, d^j)$$
$$\nexists C^i \in P\ s.t.\ \forall j <= i,\ (e^j \in C\ \wedge\ d^j \notin C)\ \vee\ (e^j \notin C\ \wedge\ d^j \in C)$$

then the set of observable behaviors of Program P' is a subset of program P.

Proof. The proof for this is fairly striaghtforward.

Condition 1 corresponds to Theorem 5.1. From Prop 3, it is sufficient to show Condition 1 to represent all events e^i, d^i in a loop iteration. (rephrase the last sentence)

Condition 2 and 3 correspond to Corollary 5.1.1 with a slight difference. Because we reorder e and d within a loop, the resultant program's Candidates C'^i will have for each iteration of the loop the events e and d reordered within them. Hence, we need to ensure that reordering is possible in every possible iteration. Condition 2 and 3 is precisely the set of conditions where we can assure that such a reordering is possible in any iteration of the loop.

Since the compiler cannot practically check for all iterations the set of conditions we have, one might assume that this does not hold in practice. On the contrary, its practical application would just involve checking Reord(e,k) and $Reord(k, d)$ for all such events k that can exist between e and d. This set can be obtained using a striaghtforward flow analysis. Additionally, Condition 3 can always be checked beforehand as it corresponds to checking whether events e and d belong in different conditional branches.

Discuss the proof with Clark.

□

Loop invariant code motion What is not so obvious is the case when events are reordered out of the loop. We will not construct a proof for this, rather use a direct counterexample to show our point.

Show figure here.

Explain figure here.

The problem is that we cannot use the notion of reordering of events in a Candidate to generate the candidate that is intended. Reordering at the Candidate level does not map directly to Reordering that is done to perform something like Loop invariant code motion.

We will later show int he next chapter that we can infact define one of the forms of loop invariant code motion using both Reordering and Elimination at the Candidate level.

To summarize, this chapter addressed the validity of instruction reordering under the ECMAScript Memory Model. We first built a conservative proof for reordering based on candidate executions. We later extended it to programs abstracted to the set of shared memory events. We discussed throughout the limitation and advantages of our conservative approach. We also presented examples throughout this chapter to get a fair intuitive understanding of the ideas behind the proof and the role of the axiomatic model in it.

In the next chapter, we will address the validity of elmination udner the ECMAScript Memory Model.

Chapter 6

Elimination

6.1 Elimination

There are two types of elimination we are concerned with:

- Read Elimination
- Write Elimination

Theorem 6.1. *Consider a candidate C of a program and its possible Candidate Executions where $\overrightarrow{hb}$ 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 $e \overrightarrow{rf} w$ 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 $\overrightarrow{hb}$ 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) $k \overrightarrow{hb} e$ b) $e \overrightarrow{hb} k$

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

$$K_b = \{k \mid k \overrightarrow{hb} e\}.$$
$$K_a = \{k \mid e \overrightarrow{hb} k\}.$$

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

We need to ensure the following relations hold after elimination.

$$\forall k_a \in K_a \ \wedge \ \forall k_b \in K_b \ . \ k_b \ \overrightarrow{hb} \ k_a \tag{6.1}$$

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

$$\forall k_b \neq p_b \in K_b \ . \ k_b \ \overrightarrow{hb} \ p_b \tag{6.2}$$

$$\forall k_a \neq p_a \in K_a \ . \ p_a \ \overrightarrow{hb} \ k_a \tag{6.3}$$

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:

$$k \ \overrightarrow{hb} \ e \ \vee \ e \ \overrightarrow{hb} \ k \tag{6.4}$$

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 $\overrightarrow{hb}$ relations are introduced, and because original candidate executions have $\overrightarrow{hb}$ 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 $\overrightarrow{rf}$ 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) $k \ \overrightarrow{hb} \ R_u o$

b) $R_u o \ \overrightarrow{hb} \ k$

In the first case, we have the following possibilities.

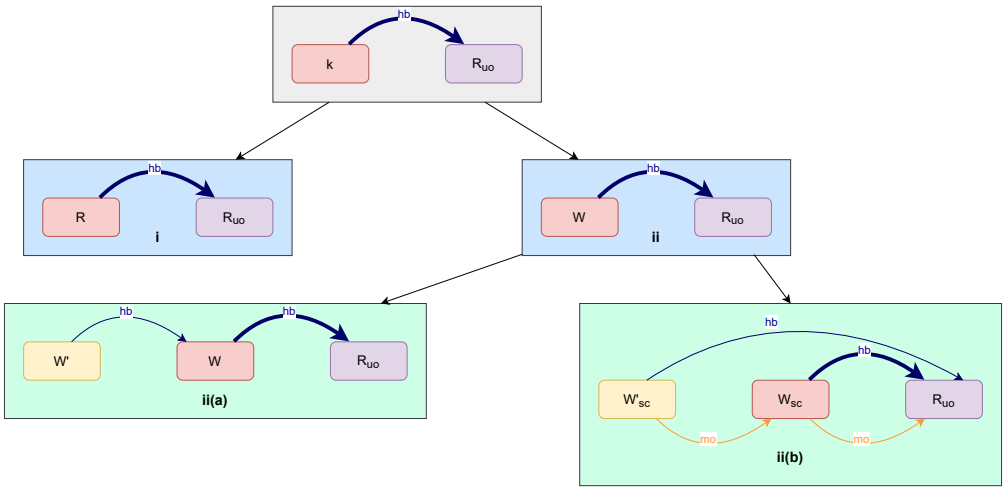


Figure 6.1: The first type of relations removed and the various patterns forbidden by them.

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.

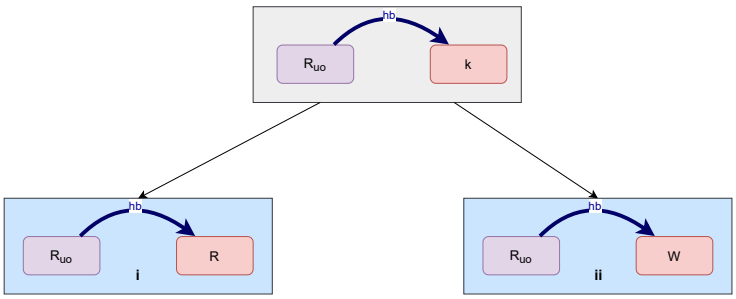


Figure 6.2: The second type of relations removed and the various patterns forbidden by them.

Observations:

- (i) is not a pattern in any Consistency rules
- (ii) is a pattern in Coherent Reads, however, only restricting $\overrightarrow{rf}$ 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 $\overrightarrow{hb}$ relations.

□

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

Theorem 6.2. *Consider a candidate C of a program and its possible Candidate Executions where $\overrightarrow{hb}$ is strictly partial order. Consider two* ***write*** *events e and d such that cons(e, d) is true in C and $e \overrightarrow{ao} d$. 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) $k \overrightarrow{hb} e$ b) $e \overrightarrow{hb} k$

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

$$K_b = \{k \mid k \overrightarrow{hb} e\}.$$
$$K_a = \{k \mid e \overrightarrow{hb} k\}.$$

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

We need to ensure the following relations hold after elimination.

$$\forall k_a \in K_a \ \wedge \ \forall k_b \in K_b \ . \ k_b \overrightarrow{hb} k_a \tag{6.5}$$

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:

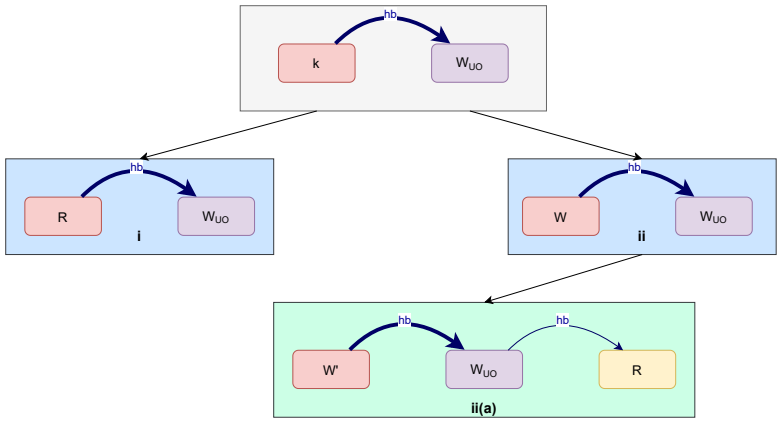


Figure 6.3: First case possibilities (change caption stimiar to that for read elim)

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:

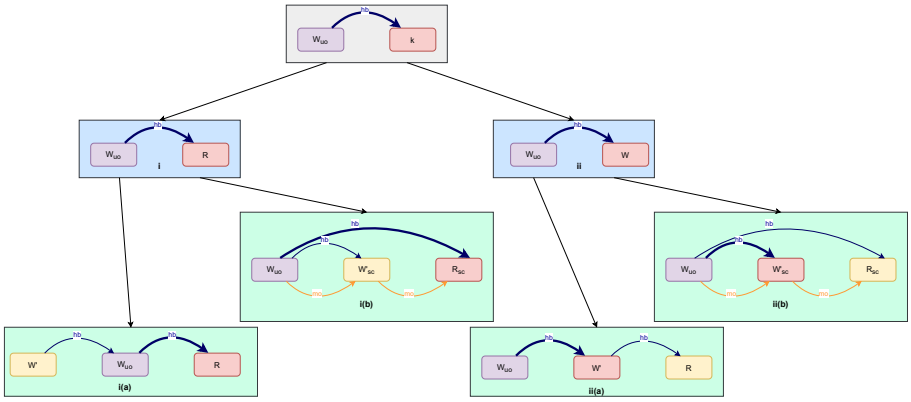


Figure 6.4: Second case possibilities (change caption stimiar to that for read elim)

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.

□

Corollary 6.2.1. *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:

$$e \overrightarrow{_{ao}} k_1 \ \wedge \ relnk_1aod$$

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 :

$$k_1 \overrightarrow{_{ao}} e \ \wedge \ e \overrightarrow{_{ao}} d$$

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

$$k_1 \overrightarrow{_{ao}} d$$

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 $\overrightarrow{_{ao}}$ 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

$$e \overrightarrow{_{ao}} k_1 \overrightarrow{_{ao}} ... \overrightarrow{_{ao}} k_{n+1} \overrightarrow{_{ao}} d \ \wedge \ cons(e, k_1) \wedge cons(k_1, k_2) \wedge ... \wedge cons(k_{n+1}, d)$$

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

The way we approach this is to not have any assumptions as to why the compiler chooses to do a particular reordering in the program. We instead only check if the reoredered program can have its observable behaviors as a subset of the original. This ensures that the algorithm for the compiler optimization need not change, but that our set of conditions will just be additional checks that can be done before actually doing the reordering. Such an approach makes reordering parametric to the memory model.

The downside is that this approach will be conservative as we use no information as to why a particular set of events are reordered. We do not compare and contrast in details the perks of both approaches. This is beyond the scope of this thesis.

6.2.1 Addressing Programs with Conditionals

We first consider the elimination of write in programs with conditional branches. The following corollary states when doing such an elimination is safe:

Corollary 6.2.2. *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 1 and Prop 1, 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

  If e and d are part of different branches of the same conditional, then we have $\neg e \overrightarrow{_{ao}} d$. Hence this case need not be considered.

  There are remaining four types of this case that we need to consider:

  1. Type 1:

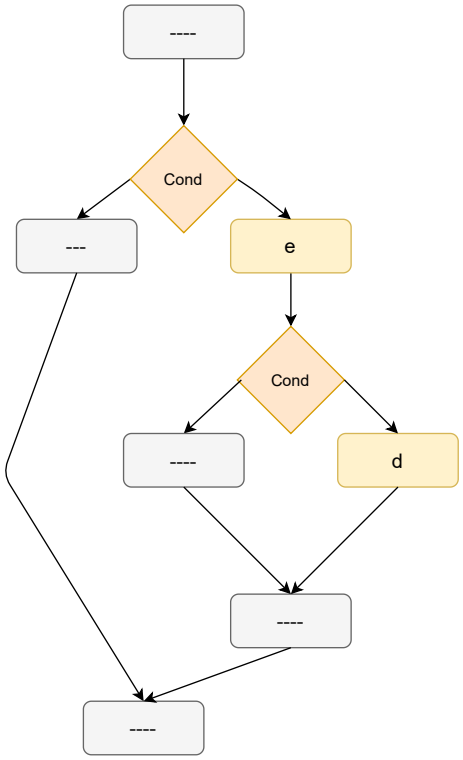


Figure 6.5: Type 1:

From the figure, we can infer by Prop 1 that

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

After elimination e, we can have a new observable behavior in a candidate not having d from the above property of this case.

2. Type 2:

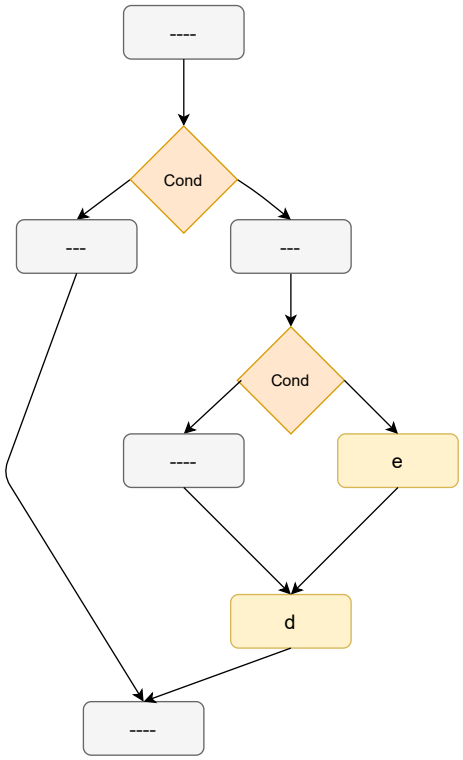


Figure 6.6: Type 2:

From the figure, we can infer by Prop 1 that

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

After elimination e, we cannot have any new observable behavior in a candidate not having d as by Prop 1, we have for the original program that

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

3. Type 3:

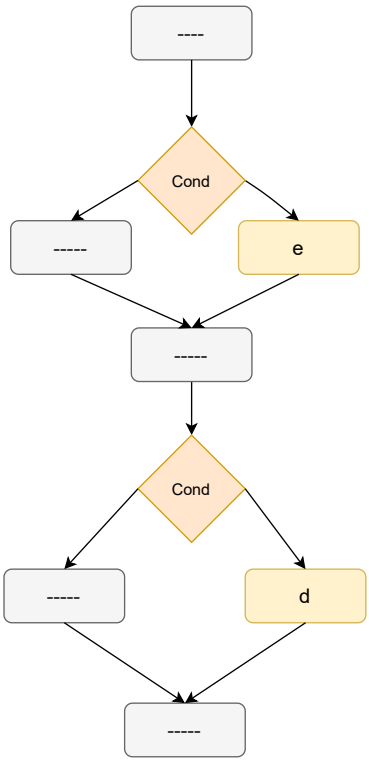


Figure 6.7: Type 3:

If e and d are part of different conditional branches, then by Prop 2 and 1, 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.

4. Type 4:

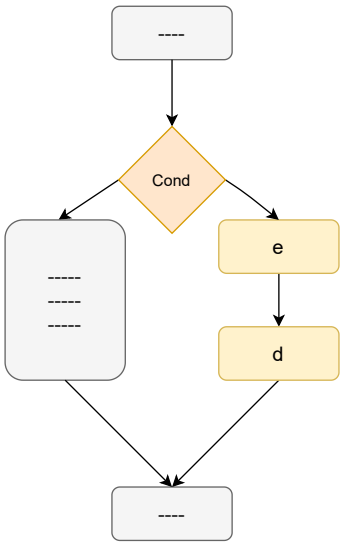


Figure 6.8: Type 4:

From the figure, we can infer by Prop 1 that

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

After elimination e, we cannot have any new observable behavior in a candidate not having d as we have by Prop 1 that

$$e \in C \ \Rightarrow \ d \in C$$

Not sure how to come to the above conclusion apart from the fact that its obvious.

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

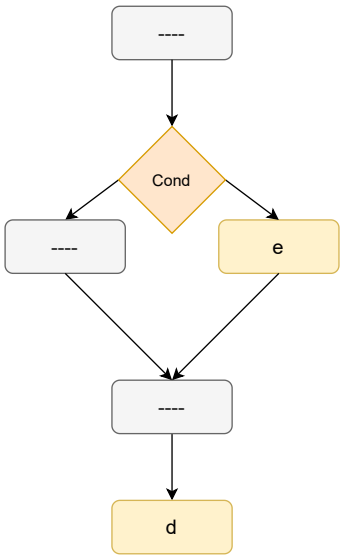


Figure 6.9:

By Prop 2 and 1, 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

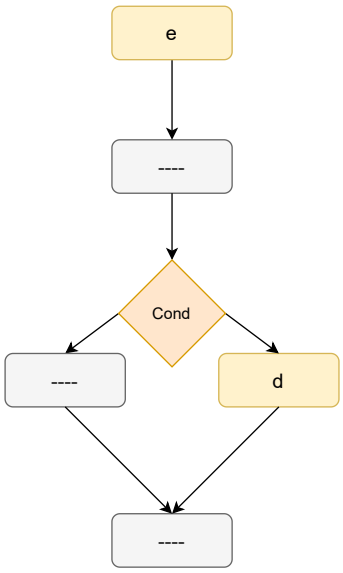


Figure 6.10: 4:

By Prop 2 and 1, 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. But what is the resultant code after elimination relies on the intention of the compiler, which can be the following:

- It could be plain dead code elimination, wherein both brnaches of code are eliminated entirely.
- It could also be that the conditonal check always returns the same value, which makes the branch taken to be the same.
- It could also be that the choice of branch does not affect the outcome of the program itself.

Since we aren't certain of the reason, it is difficult to identify the target code that is inteneded after such an elminiation. Hence we do not address this case. It is also not within the scope of our analysis to conhsder the actual mapping between program and candidates. We would need this to prove that the program does not take a particular conditonal branch in any execution. This is not easy to do without the mapping in our hands.

6.2.2 Addressing Programs with Loops

Similar to reordering, for the sake of elimination, we consider programs with just one loop.

We first consider the simpler case of read elimination within a loop. Eliminating such a read at a program level would imply in every candidate C^i where i denotes the number of iterations, the read R within the loop can be eliminated.

By Thereom 1 of read elimination, we only need the read R to have the type uo to perform such an elimination. Thus we can eliminate for each iteration of the loop our intended read. By transitive property of subsets, the resultant candidate will have observable behaviors as a subset of the original. Doing this for all valid canddiates extends to program level.

Next we consider the case of eliminating writes within a loop. Eliminating such a read at a program level would imply in every candidate C^i where i denotes the number of iterations, the read W within the loop can be eliminated.

For this, we need to have some write d that can exist in all candidates where e can exist. ($\nexists C \in P$ s.t $e \in C \wedge d \notin C$). This condition corresponds to Corollary 2 of elimination, which handles the case of conditonals too. Next, we need to show that in each iteration the write e can be eliminated. We once again have Corollary 2 to show when we can do this. By transitive property of subsets, we can show that the resultant candidate has observable behaviors as a subset of original. Doing this for all valid canddiates extends to program level.

Loop Invariant Code motion In the previous chapter, we showed that loop invariant code motion cannot be validated by just reordering at the Candidate level. This was because reordering was insufficient to generate the resultant candidate from the original. This however can be done by coupling Reordering with Elimination.

We first consider the case of reordering a Read outside a loop.

Corollary 6.2.3. *Consider K to be the set of events within a loop in program P. Consider e to be a read within the loop. Consider program P' with event e agent ordered before the loop. If*

$$e : uo$$
$$\forall k \neq e \in K,\ Reord(k, e)$$
$$\nexists C^i \in P \ s.t.\ e^{j<=i} \notin C$$

then the set of observable behaviors of P' is a subset of P.

Proof. We first consider the program with just one iteration. Hence for Candidate C^1, we have just e^1. We need to ensure that the resultant candidate C'^1 such that

$$\forall k \in K,\ e \xrightarrow[ao]{} k$$

has observable behaviors as a subset of C^1

For every $k \in K$ such that $k \xrightarrow[ao]{} e$ we have from Condition 2 $Reord(k, e)$. Condition 3, from Prop 1 implies that e is not part of any conditional branch. Thus, from Corollary 5.1.2 and Corollary 5.1.4, we can infer that C'^1 has observable behaviors as a subset of C^1.

Next, we consider the program with more than one iteration of the loop. We prove this case using induction on the number of reads e that exist due to multiple iterations of the loop.

- Base case : number of $e = 2$

  This case corresponds to candidates of the form C^2, thus giving us two reads e^1 and e^2. From Condition 2, we have

  $$\forall k \in K,\ Reord(k, e^1) \wedge Reord(k, e^2)$$

  From Corollary 5.1.2 and Corollary 5.1.4, we can infer that C''^2 has observable behaviors as a subset of C^2. To go from C''^2 to C'^2, note that in C''^2 we have $cons(e^1, e^2)$ after reordering them. From Theorem 6.1, we can eliminate either e^1 or e^2, thus resulting in C'^2 whose observable behaviors is a subset of C''^2.

  By transitive property of subsets we can infer that C'^2 has observable behaviors as a subset of C^2.

- Inductive case : number of $e =$ n

  Assume that for all such candidates with n iterations of the loop, the observable behaviors of C'^n is a subset of C^n.

  We now prove using this that it can also hold for number e as $n + 1$. This case corresponds to candidates of the form C^{n+1}, thus giving us $n + 1$ reads $e^1, e^2, ..., e^{n+1}$. From Condition 2, we have

  $$\forall k \in K,\ Reord(k, e^{n+1})$$

  using which we can infer

  $$\forall k \text{ s.t. } e^n \xrightarrow[ao]{} k \wedge k \xrightarrow[ao]{} e^{n+1},\ Reord(k, e^{n+1})$$

  From Corollary 5.1.2 and Corollary 5.1.4, we can infer that C''^{n+1} with $cons(e^n, e^{n+1})$ due to reordering e^{n+1} has observable behaviors as a subset of C^{n+1}. By Theorem 6.1, we can eliminate e^{n+1}, thus giving us candidate of the form C^n whose observable behaviors is a subset of C''^{n+1}.

  From our inductive assumption, we can then conclude that C'^{n+1} has observable behaviors as a subset of C^n. By transitive property of subsets, we can infer that C'^{n+1} has observable behaviors as a subset of C^{n+1}.

  Mind the notations. Perhaps have another reivew of it to avoid confusion of the reader.

  The argument using Theorem of Read elimination is somewhat cheeky, since we have $cons(e^i, e^{i+1})$ where the two reads are to the same local variable. This additional condition may have to be put in Read Elimination. But let us discuss that with Clark later.

□

Corollary 6.2.4. *Consider K to be the set of events within a loop in program P. Consider e to be a read within the loop. Consider program P' with event e agent ordered before the loop. If*

$$e : uo$$
$$\forall k \neq e \in K,\ Reord(e, k)$$
$$\nexists C^i \in P \text{ s.t. } e^{j<=i} \notin C$$

then the set of observable behaviors of P' is a subset of P.

Proof. We first consider the program with just one iteration. Hence for Candidate C^1, we have just e^1. We need to ensure that the resultant candidate C'^1 such that

$$\forall k \in K,\ k \xrightarrow[ao]{} e$$

has observable behaviors as a subset of C^1

For every $k \in K$ such that $e \xrightarrow[ao]{} k$ we have from Condition 2 $Reord(e, k)$. Condition 3, from Prop 1 implies that e is not part of any conditional branch. Thus, from Corollary 5.1.3 and Corollary 5.1.4, we can infer that C'^1 has observable behaviors as a subset of C^1.

Next, we consider the program with more than one iteration of the loop. We prove this case using induction on the number of reads e that exist due to multiple iterations of the loop.

- Base case : number of $e = 2$

  This case corresponds to candidates of the form C^2, thus giving us two reads e^1 and e^2. From Condition 2, we have

  $$\forall k \in K,\ Reord(e^1, k) \wedge Reord(e^2, k)$$

  From Corollary 5.1.2 and Corollary 5.1.4, we can infer that C''^2 has observable behaviors as a subset of C^2. To go from C''^2 to C'^2, note that in C''^2 we have $cons(e^1, e^2)$ after reordering them. From Theorem 6.1, we can eliminate either e^1 or e^2, thus resulting in C'^2 whose observable behaviors is a subset of C''^2.

  By transitive property of subsets we can infer that C'^2 has observable behaviors as a subset of C^2.

- Inductive case : number of e = n

  Assume that for all such candidates with n iterations of the loop, the observable behaviors of C'^n is a subset of C^n.

  We now prove using this that it can also hold for number e as $n+1$. This case corresponds to candidates of the form C^{n+1}, thus giving us $n+1$ reads $e^1, e^2, ..., e^{n+1}$. From Condition 2, we have

  $$\forall k \in K,\ Reord(e^n, k)$$

  using which we can infer

  $$\forall k \text{ s.t. } e^n \xrightarrow[ao]{} k \wedge k \xrightarrow[ao]{} e^{n+1},\ Reord(e^n, k)$$

  From Corollary 5.1.2 and Corollary 5.1.4, we can infer that C''^{n+1} with $cons(e^n, e^{n+1})$ due to reordering e^{n+1} has observable behaviors as a subset of C^{n+1}. By Theorem 6.1, we can eliminate e^{n+1}, thus giving us candidate of the form C^n whose observable behaviors is a subset of C''^{n+1}.

  From our inductive assumption, we can then conclude that C'^{n+1} has observable behaviors as a subset of C^n. By transitive property of subsets, we can infer that C'^{n+1} has observable behaviors as a subset of C^{n+1}.

  Mind the notations. Perhaps have another reivew of it to avoid confusion of the reader.

  The argument using Theorem of Read elimination is somewhat cheeky, since we have $cons(e^i, e^{i+1})$ where the two reads are to the same local variable. This additional condition may have to be put in Read Elimination. But let us discuss that with Clark later.

□

Now we consider the case of reordering a write outside a loop:

Corollary 6.2.5. *Consider K to be the set of events within a loop in program P. Consider e to be a read within the loop. Consider program P' with event e agent ordered before the loop. If*

$$e : uo$$
$$\forall k \neq e \in K,\ Reord(k, e)$$
$$\nexists C^i \in P \text{ s.t. } e^{j<=i} \notin C$$

then the set of observable behaviors of P' is a subset of P.

It is interesting to note that we do not need Reord(e,k) which was originally our plan because we needed to eliminate every e^j w.r.t e^{j+1}. Because we have Reord(k,e), we can get all the writes to be consecutive to each other. Thus by Theorem 1 directly, we can eliminate them all. We do not require Corollary of Elimination here.

Proof. We first consider the program with just one iteration. Hence for Candidate C^1, we have just e^1. We need to ensure that the resultant candidate C'^1 such that

$$\forall k \in K,\ e \xrightarrow[ao]{} k$$

has observable behaviors as a subset of C^1

For every $k \in K$ such that $k \xrightarrow[ao]{} e$ we have from Condition 2 $Reord(k,e)$. Condition 3, from Prop 1 implies that e is not part of any conditional branch. Thus, from Corollary 5.1.2 and Corollary 5.1.4, we can infer that C'^1 has observable behaviors as a subset of C^1.

Next, we consider the program with more than one iteration of the loop. We prove this case using induction on the number of reads e that exist due to multiple iterations of the loop.

- Base case : number of $e = 2$

  This case corresponds to candidates of the form C^2, thus giving us two reads e^1 and e^2. From Condition 2, we have

  $$\forall k \in K,\ Reord(k,e^1) \wedge Reord(k,e^2)$$

  From Corollary 5.1.2 and Corollary 5.1.4, we can infer that C''^2 has observable behaviors as a subset of C^2. To go from C''^2 to C'^2, note that in C''^2 we have $cons(e^1,e^2)$ after reordering them. From Theorem 6.2, we can eliminate e^1, thus resulting in C'^2 whose observable behaviors is a subset of C''^2.

  By transitive property of subsets we can infer that C'^2 has observable behaviors as a subset of C^2.

- Inductive case : number of $e =$ n

  Assume that for all such candidates with n iterations of the loop, the observable behaviors of C'^n is a subset of C^n.

  We now prove using this that it can also hold for number e as $n+1$. This case corresponds to candidates of the form C^{n+1}, thus giving us $n+1$ reads $e^1, e^2, ..., e^{n+1}$. From Condition 2, we have

  $$\forall k \in K,\ Reord(k,e^{n+1})$$

using which we can infer

$$\forall k \text{ s.t. } e^n \xrightarrow[ao]{} k \wedge k \xrightarrow[ao]{} e^{n+1},\ Reord(k, e^{n+1})$$

From Corollary 5.1.2 and Corollary 5.1.4, we can infer that C''^{n+1} with $cons(e^n, e^{n+1})$ due to reordering e^{n+1} has observable behaviors as a subset of C^{n+1}. By Theorem 6.2, we can eliminate e^{n+1}, thus giving us candidate of the form C^n whose observable behaviors is a subset of C''^{n+1}.

From our inductive assumption, we can then conclude that C'^{n+1} has observable behaviors as a subset of C^n. By transitive property of subsets, we can infer that C'^{n+1} has observable behaviors as a subset of C^{n+1}.

Mind the notations. Perhaps have another reivew of it to avoid confusion of the reader.

□

Corollary 6.2.6. *Consider K to be the set of events within a loop in program P. Consider e to be a read within the loop. Consider program P' with event e agent ordered before the loop. If*

$$e : uo$$
$$\forall k \neq e \in K,\ Reord(e, k)$$
$$\nexists C^i \in P \text{ s.t. } e^{j<=i} \notin C$$

then the set of observable behaviors of P' is a subset of P.

It is interesting to note that we do not need $Reord(e, k)$ which was originally our plan because we needed to eliminate every e^j w.r.t e^{j+1}. Because we have $Reord(k, e)$, we can get all the writes to be consecutive to each other. Thus by Theorem 1 directly, we can eliminate them all. We do not require Corollary of Elimination here.

Proof. We first consider the program with just one iteration. Hence for Candidate C^1, we have just e^1. We need to ensure that the resultant candidate C'^1 such that

$$\forall k \in K,\ k \xrightarrow[ao]{} e$$

has observable behaviors as a subset of C^1

For every $k \in K$ such that $k \xrightarrow[ao]{} e$ we have from Condition 2 $Reord(e, k)$. Condition 3, from Prop 1 implies that e is not part of any conditional branch. Thus, from Corollary 5.1.2 and Corollary 5.1.4, we can infer that C'^1 has observable behaviors as a subset of C^1.

Next, we consider the program with more than one iteration of the loop. We prove this case using induction on the number of reads e that exist due to multiple iterations of the loop.

- Base case : number of $e = 2$

  This case corresponds to candidates of the form C^2, thus giving us two reads e^1 and e^2. From Condition 2, we have

  $$\forall k \in K,\ Reord(e^1, k) \wedge Reord(e^2, k)$$

  We also have from property of loops that $e^1 \xrightarrow[ao]{} e^2$. From Corollary 6.2.1, we can eliminate e^1, giving us C''^2 whose observable behaviors as a subset of C^2. From Corollary 5.1.3 and Corollary 5.1.4, we can reorder e^2 outside the loop thus giving us C'^2 whose observable behaviors as a subset of C''^2.

  By transitive property of subsets we can infer that C'^2 has observable behaviors as a subset of C^2.

- Inductive case : number of $e =$ n

  Assume that for all such candidates with n iterations of the loop, the observable behaviors of C'^n is a subset of C^n.

  We now prove using this that it can also hold for number e as $n + 1$. This case corresponds to candidates of the form C^{n+1}, thus giving us $n + 1$ reads $e^1, e^2, ..., e^{n+1}$. From Condition 2, we have

  $$\forall k \in K,\ Reord(e^n, k)$$

  using which we can infer

  $$\forall k \text{ s.t. } e^n \xrightarrow[ao]{} k \wedge k \xrightarrow[ao]{} e^{n+1},\ Reord(e^n, k)$$

  From Corollart 6.2.1, we can eliminate e^n, thus giving us candidate C^n whose observable beahviors are a subset of C^{n+1}

  From our inductive assumption, we can then conclude that C'^{n+1} has observable behaviors as a subset of C^n. By transitive property of subsets, we can infer that C'^{n+1} has observable behaviors as a subset of C^{n+1}.

  Mind the notations. Perhaps have another reivew of it to avoid confusion of the reader.

  Proof read later.

□

Reordering two events accross loops Note that we still cannot assert when we can reorder events inside a loop. This would require a proof of redundancy introduction at the candidate level. This is beyond the scope of this thesis.

Perhaps you can elaborate a bit more on this.

Chapter 7

Conclusion, Summary, Future Work

The previous chapter addressed the validity of elimination of relaxed memory accesses. We also showed how validity of loop invariant code motion can be done using reordering coupled with elimination of events. In this concluding chapter, we discuss the limitations of our approach from a practical standpoint. We later chart out further steps that can be taken from our work that we find important to address. We elicit takeaway from this thesis work and conclude with possible future work in the domain of relaxed memory models.

7.1 Limitations/Advantages

7.1.1 Separation of Concerns

1. While our approach to program transformations avoids the operational complexity of the language, it also takes no information about the reason certain program transformations are done by the compiler/ hardware.

2. The compiler might have more information about the validity of a program transformation sequentially, which we do not take into account, while proving whether it is valid under weak memory.

3. As a simple example, it might turn out to be that sequentially, a conditional will always return true. Hence the compiler might choose to reorder the events within the *true* branch outside.

4. But as per our corollaries, we do not allow any reordering outside the loop, simply because we have no such information about the fact that a conditional always returns the same value. Having such information can give us more fine grained analysis of when reordering is valid.

5. The reason we did not consider the reasons why the compiler would do such a thing is only due to the sheer complexity of whole program analysis. There might be information which comes due to inter/intra-procedural analysis of programs, which may account for a very large number of cases to address the validity of reordering.

6. Hence, in our approach, we assume that the compiler knows what it is doing sequentially and do not assert any implicit assumption as to why the compiler would do a program transformation.

7. This results in a clear separation of concerns, and though our results may not be directly applied in practice, we believe it gives the compiler writer enough information from a relaxed memory perspective to carefully design the algorithms for program transformations.

7.1.2 Validity of Transformations is Sound but not Complete

1. It is paramount to note that our approach to validity of elimination and reordering is conservative.

2. We do not assume anything about events that belong to other agents/threads. Rather, we only use information on the events involved in the program transformation and those that are *agent-ordered* between them.

3. Such a conservative approach gave us the situation when reordering is safe, hence our approach is sound. But it is not optimal, in the sesne, it is not complete while considering the entire program. There are several cases where one could reorder or eliminate events that we prohibit, but are still safe to do.

4. This, however, as mentioned before, is to specific programs. To keep track of such information while doing program transformation in practice is infeasible as the program size increases.

5. We wanted to ensure that the compiler can use minimal local information to assert validity of doing a program transformation, while also allowing most cases where it is in general safe to do. (rephrase)

6. This by chance turned out to be a good approach as we could use lemmas over partial order relations (like happens-before) to reason about validity of transformations at a global level using jsut some local information.

7. This exemplifies the power of reasoning about relaxed memory semantics using an axiomatic specification.

7.1.3 Lack of Practical Results

1. This work is purely theoretical. There is yet much more to be done to extend our results into practice.

2. Literature has shown that mere empirical testing of results on concurrent programs is insufficient.

3. One must have a theoretical proof as to why certain results hold in a concurrent setting.

4. The same is for the case of relaxed memory.

5. Extending our results into practice is beyond the scope of a Master's thesis.

7.1.4 Mapping from Programming Constructs to Abstract events

1. The specification and the results on program transformations are purely at the abstract set of shared memroy accesess.

2. The concise mapping from ECMAScript's Read/Write to these abstract events is somethign that must be done.

3. This mapping however, is not required for our results, meaning, they do not influence it in any way.

4. However, to apply our results in practice, such a mapping would be required. (for example Atomics.load or Atomics.store, based on their size of read/write would imply)

5. As an example, we might have to perform aliasing analysis to identify which shared memory accesses are of same range.

7.2 Steps Further

7.2.1 Addressing Read-Modify-Write

1. So far we have assumed that no read modify write events exist in programs.
2. However, this assumption is too strong in general.
3. Analysis of the validity of reordering/elimination when RMW events are involved should be done to have a complete analysis of these two transformations as far as shared memory accesses are concerned.

7.2.2 Incorporating Tearing Factor

1. The role of tearing is still not clear to us.
2. Axiom 2 does not rely on any partial order relations other than reads-from.
3. Since our approach is mainly reliant on preserving happens-before, our intuition is that our results should ideally be independant of the tearing factor.
4. However, a proof including tearing aspect for each event is still needed.

7.2.3 Role of synchronize/host-specific events

1. We have not yet considered the role of synchronize events.
2. Though for a programmer this is equivalent to wait and notify, reordering and elimination under their presence is something we have not conisdered.
3. This we suspect would require understanding the operational aspect of wait / notify procedures.

1. We do not yet know how Host Specific synchronize events work with relaxed memory acceses.
2. Strictly speaking, the semantics from a consistency model perspective is same as that of synchronize events.
3. However, a detailed analysis must be done before incorporating its role.

7.2.4 Addressing other basic program transformations

1. Addressing redundancy introduction as an immediate next step would prove useful.

2. Using it, we can analyze reordering of events accross loops.

3. This will also give an interesting equivalence to instruction reordering.

4. Other program transformations we find important to consider are strengthening/ weakening access modes (fence optimizations), gathering optimization, changing tearing factor of accesess.

7.3 Critique of the Model itself

7.3.1 Range of Initialize events uncertain

7.3.2 Unordered events do not respect Coherence

7.3.3 Out of Thin air values

7.3.4 Sequentially Consistent events can be in a data race

7.4 Takeaway from this work

7.4.1 Addressing Concurrency Problems

7.4.2 Separation of Concerns

7.4.3 Weak Memory Models still ill understood

7.4.4 Addressing validity of Program Transformations in Concurrent Context

7.5 Future Directions in Weak Memory Consistency

7.5.1 Specification of Mixed-Size memory models

Most of the work done towards addressing concerns of memory model relied on the assumption that shared memory accesses are all of the same size. However, hardware does allow accesses of multiple sizes and while looking at this from a relaxed memory access standpoint, the semantics of such accesses is still quite unclear. Part of it is due to hardware vendors not able to decide on what kind of behaviors they want to allow for programs using such accesses, which brings another concern for high-level programming languages; describing semantics to use mixed size accesses becomes difficult.

THis thesis is based on one such model and its impact on program transformations. But this model was quite simple in that the aspect of mixed-size and their behavior for accesses such as atomic were semantically not defined. This may not be the case for hardware models such as ARMv8. In my literature review in this

direction, there is still not a well understood memory model, let alone much work done on impact of such models on program transformations.

One direction to go is to have a concise analysis of the validity of program transformation under mixed-size models such as ARMv8 and the new possible transformations that come along with them (like merging two accesses or splitting an access into multiple accesses.) To do this would also require formally describing the mixed size model (if haven't already, for example check [**?**] for an older version of ARM model), which I also consider worthwhile.

7.5.2 Transformational Specification of Memory Models

Using relaxed memory accesses certainly has a good impact on performance. However, as we see in this thesis, their semantics shown to be often not so suitable for program transformations done by the compiler or the hardware. This work addressed a small part of this problem, by formalizing one such weak memory model and constructing a proof to show when a few basic program transformations are valid to perform.

However, over reading literature on work done in this way, it has come to my knowledge that the validity of program transformations still remain a problem. As new concurrent languages come, new memory models are introduced and the validity of program transformations have to be addressed for each such model separately. Instead, one way to consider going about is by describing them formally using program transformations. In a very preliminary literature survey in this direction, only one reserch paper [**?**] that tries to do this. However, it is still quite limited. Given the different memory models that exist today for many concurrent languages and hardware, having a formal model for the well known memory models would in our perspective be a good start. It would prove useful for designing new memory models, compilers as well as understanding them from a programmer's standpoint.

7.5.3 Automation of Specification of Weak Memory Models

This thesis also showcased some counter-examples to show that some transformations may not be safe. In standard literature, such an approach is also used to explain or identify loopholes in specification of memory models. Known as litmus tests, they prove useful to identify which features of weak-memory consistency does a given model have.

However, there is little work done in inferring specifications themselves using such litmus tests. The problem is that most often it becomes difficult to concisely describe

a model as a set of litmus tests. While a lot of work has been done in identifying key examples as litmus, the work in direction of mixed-size models has just begun.