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

With the increasing demand for performance in computing, the use of concurrency has become quite ubiquitous. Within this, the use of shared memory concurrency in contrast to messgae passing has shown to give tremendous performance benefits. A big part of this is due to using relaxed memory accesses, which sacrifice the "atomicity" of actions to gain high speedups.

The use of such accesses, unmonitored can actually cause programs to mis-behave or can cause quite a bit of slowdown. This is due to the lack of well defined semantics that can guide the user to utilize these accesses more responsibly. The lack of specification also may invalidate several aggressive opimtizations that the compiler does which contributes to a major chunk of our so called "performance".

In current times, there is an increasing trend / necessity to off load computations to the web. One major benefit of this is that portability issues are reduced greatly, which means I can run a complex software and use it on my browser even though my system is not compatible to run it. To ensure that computations done on the web be competible with those done offline, performance plays a major factor, and this factor has always pushed people to not rely on web-based compuatations.

In this scenario, the use of shared memory access (more specifically, relaxed accesses) naturally fits in to provide hte lack of performance we so desperately need. Recently, ECMAScript, the standard for all such web-scriptuing languages defined their own memory consistency model. This model, however, is written in prose format with no formal proof about the validity of common program transformations. In addition, this model also described semantics for mixed-size accesses, something which is quite recent and does not exist for standard memory models.

Our focus, therefore, in this thesis is to first 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 an intuitive, conservative proof of when reordering and elimination of instructions is permitted, addressing optimization in terms of its impact on observable program behaviours.

We do the above by first giving a Declarative (commonly known as Axiomatic) specification of the ECMAScript memory model. This model is described using partial orders between events that constitute our program. We use these partial orders to define a program execution and its outcomes(we refer to as Observable Behaviors). The axioms of the model help us define which of these outcomes are valid.

We then use this axiomatic formulation to reason about reordering and elimination at the execution level of program. We then move towards program level by reasoning about these transformations in the presence of conditionals and loops. Throughout this process, we use examples wherever required to give a better intuitive understanding of the proofs as well as the axioms of the model.

We conclude with the limitations/advantages of our approach. We also give further steps that can be taken to address such limitations as well as pending work that could be done. Later, we critique the model using some examples, hinting towards underspecification in certain parts. We end by exposing certain foundational problems/directions that we identified in the process of doing this thesis which in our eyes need to be addressed in the immediate future.

Specific contributions of our work include the following:

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

2. Using this model, we show when basic reordering of independent instructions is allowed. We extend this to reordering in the presence of conditionals and loops in programs.

3. Similar proof designs are used to validate other basic optimization behaviours such as removing redundant reads or writes. Further extending it to elimination in the presence of conditionals and loops.

4. We lastly show how our above two results help us check the validity of loop invariant code motion.

This thesis includes 5 more chapters, which organize the above contributions as follows:

- Chapter 2 gives a *background* to this topic which are coupled with relevant related work.
- Chapter 3 give the formal *declarative style* semantics of the ECMAScript memory model.
- Chapter 4 dives into *instruction reordering*, with formulation of relevant *lemmans, theorems and corollaries* (with appropriate proofs) that show when reordering is valid.
- Chapter 5 does the same as the previous chapter for *instruction elimination*. This chapter also gives a proof for validity of *loop invariant code motion*.
- Chapter 6 concludes by eliciting the *limitations* of our approach, our *critique of the model* and further steps that can be taken to address pending problems (future work).

Clark Could you please edit this yourself? I do not know how to write this properly. Part of the problem is should I write this in first person or third person form.

The formal semantics of the model and the proof for reordering at the execution level was published in the Workshop on Languages and Compilers for Parallel Computing (LCPC) 2020 **Akshay**. Akshay Gopalakrishnan is the first author of the paper. Clark Verbrugge is the second as well as the thesis supervisor.

Chapter 2

Background

This chapter starts with a review of key previous work done in the domain of relaxed memory models. We start by eliciting research works done in the design of memory models, coupled with works exposing problems due to ill defined or informally specified semantics. We then elicit research works done in the context of different memory models with respect to validity of program transformations. Finally, we end with a list of tutorial works done in the axiomatic style specification of memory models.

2.1 Memory Consistency Models

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*.

A lot of problems emerged with the introduction of concurrent computing. Starting with the famously known mutual exclusion problem way back in the 1960s, Djikstra et.al [2] introduced this problem and a solution to it. From it stemmed the ideas of semaphores and monitors that we all know today. Since then, a lot of well known problems in concurrency such as The Producer Consumer Problem, Dining Philosopher's problem, Reader-Writer problem have come up, along with different algorithms to implement these in a practical sense, especially in Operating Systems.

However, the above concerns in concurrency only revolved around the so-called "critical section"; which required that every thread trying to do something in a shared memory region must have exclusive access to it before doing anything. But as years went by, the need for more performance had increased, which also resulted in multiple hardwares having differnt features such as read-write buffers, caches, etc. Reliance on purely critical sections to access perform operations on shared memory was observed as a bottleneck. Without its use, the possible outcomes of a concurrent program increased, as well as the notion of *atomicity* in actions done on shared memory became relatively unclear.

Sequential Consistency, which was first formulated by Lamport et al. [3], gives programmers a very intuitive way to reason about their programs running in a multiprocessor environment with our without the use of critical sections. *Sequential Consistency* (SC) guaranteed 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 2.1 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.

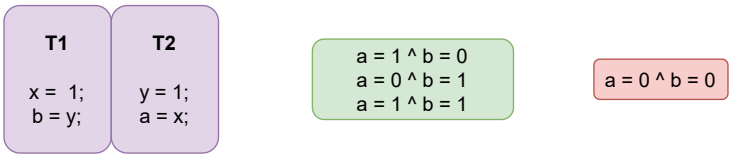


Figure 2.1: Example program with its possible outcomes under sequential consistency.

However, the above program under SC cannot have the outcome $a = 0 \ \wedge \ b = 0$ as given in the red box to the right. To show that it is indeed not possible, assume that $a = 0$. This means $x = 1$ has not been done while $y = 1$ has been done. This also means $b = y$ has not been done yet. Hence, if now the read to y occurs, it cannot be 0. The case is similar for when $b = 0$. Such sequential reasoning though easy to understand and follow while writing concurrent programs, may not be that advantageous from a performance perspective.

Practically, *Sequential Consistency* was also too "strict," in the sense that it may impede possible performance benefits of using low level optimization features, such as instruction reordering, elimination, introduction done by the hardware. A tutorial by Adve et al. [4], summarizes the most common hardware features for relaxed memory that are now available in most hardware. What this tutorial also exposed is the

difficulty in formalizing such features in a way that we can reason about our programs sanely without getting caught up in the complexity of multiple executions of our programs. Such features relaxed the sequential reasoning in programs, and hence models describing their semantics came to be known as relaxed memory models. Unsurprisingly, relaxed memory model specifications for different hardware / high level programming languages are still written in informal prose format, which lead to a number of problems in implementation [5]. A position paper by Nardelli et.al [6] summarizes these problems well. This informal specifications led to a number of inconsistencies in intention of the designer and how the programs behaved.

x86 Intel's white paper on their earlier versions of the memory model for x86 was described in their white paper (CANNOT FIND CITATION TO IT). Sarkar et al. [7] showed that the then x86 model was fairly informal, which they formalize in their x86-CC memory model. Owens et al. [8] later showed that the x86-CC model did not reflect precisely what x86 harware intended, followed by exposing more problems in the latest specification of the x86 model given in the white paper of Intel [9]. They then propose a new model, x86-TSO as a remedy to this. The series of work in this exposed repeated inconsistencies between the specification and the implementation in hardware.

While memory consistency models were initially only concerning hardware langauges, as the years went by, even high level programming languages came up with their own memory models. This however, was not an easy process; several changes and edits are still made to memory models of Java and C11 to date, due to various new concerns about informal/ ill defined specifcations that render the model useless for programmers to rely on.

Java Pugh et al. [10] showed the complexity of the initial versions of Java memory model, exposing with seemingly simplistic examples that even those involved in its inception were not sure about its specifcation. Later, Manson et al. [11] also exposed many limitations and underspecified semantics within the model, for which they proposed a new model with concise semantics. Manson et.al [12] also concisely described the later version of the model, which had complex semantics to have *out-of-thin-air* guarantees for programs with data-races. Recent works such as that done by Bender et al‘. [13], also showed us that the recent updates to the Java Memory model are still relatively unclear, which they again formalize.

C++/C11 Boehm et al. [14] exposed that the earlier version of C++ concurrency semantics was unsound, followed by proposing a new semantics for the same.

Vafeiadis et.al [15] summarized the open problems and verificaiton results of the then version of C11 memory model. Some of these problems have been addressed, while some still remain open to date. Batty et al. [16] exposed the lack of clear specifications of the then version of C11 memory model, giving a clarified, mathe,matical yet seemingly readable specification of the model. Nienhuis et al. [17] gave an operational semantics of the then C11 memory model, exposing certain limitations in terms of execution of such concurrent programs. Lahav et al. [18] exposed that the current compilation scheme of C11 concurrent programs to POWER unsound, thus proposing fixes to the memory model of C11 itself, which they call $RC11$.

Mixed-size models The above memory models were all based on the assumption that memory accesses are of equal sizes. But in practice this may not always be the case. Hardware can have from 8-bit to 64-bit or event 128-bit memory accesses that can be done either atomically or split accross different subsequent memory accesses. Investigation of semantics in this direction is fairly recent. Flur et al. [19] investigated mixed-size behaviors in Arm and POWER architechtures, also exposing new problems to address in the semantics. They also extend the current C11 memory model with some mixed-size semantics. ECMAScript, being a relatively simpler mixed-size model has also had some attention in this respect. Watt et al [20] uncovered and fixed a deficiency in the previous version of the model, repairing the model to guarantee SC-DRF.

2.2 Program Transformations under Weak Memory

Although programmers usually are responsible to write efficient programs, their performance during execution does not always depend on how well the program is written. Several compiler optimizations done, coupled with run-time optimizations by hardware play a big role in the end performance of programs. For sequential programs, a lot of well established ideas exist to enhance the performance of programs, but they do not map well to that of concurrent programs. With the introduction of relaxed memory accesses, quite a few critical program transformations responsible for huge performance gains may be unsound.

This demanded for a fundamental change in which we employ data-flow analysis to optimize programs in a concurrent context. Naumovich et.al [21] proposed one such analysis which is today commonly known as May-Happen-In-Parallel analysis. Midkiff et.al [22] proposed a unified compiler for several memory models, wherein

they proposed a new Concurrent Control Flow graph with new edges to perform powerful transformations such as Common Sub-Expression Elimination. These models however, demanded us to have more of a whole program analysis to even do basic local program transformations. The data flow graph as the number of threads increase will be infeasible to be implemented and used in common practice. (I AM UNSURE IF ANY RECENT PROPOSED MODELS HAVE ANYTHING TO SAY IN THIS RESPECT.) Another way to go about this is to go one step below and observe the transformations in terms of more basic code transformations such as instruction reordering, elimination and introduction. Since our objective in this thesis is not in terms of implementing program transformations, but checking the validity, we do not investigate further in this direction.

We start by showing using counter examples how one can show basic transformations are invalidated under a consistency model.

redPErhaps cite Jade's work here on Data FLow analysis for Weak memory.

For instance, from a program transformation standpoint, the disallowed outcome in Figure 2.1 should be possible; we can simply reorder either the two events in $T1$ or that in $T2$ as they are disjoint memory operations. But from a sequential consistency standpoint, since the outcome is not valid, it also brings with it the question whether such simple program transformations are even valid to perform. Figure 2.2 shows how after doing either one of these reorderings, an outcome invalid under SC is possible.

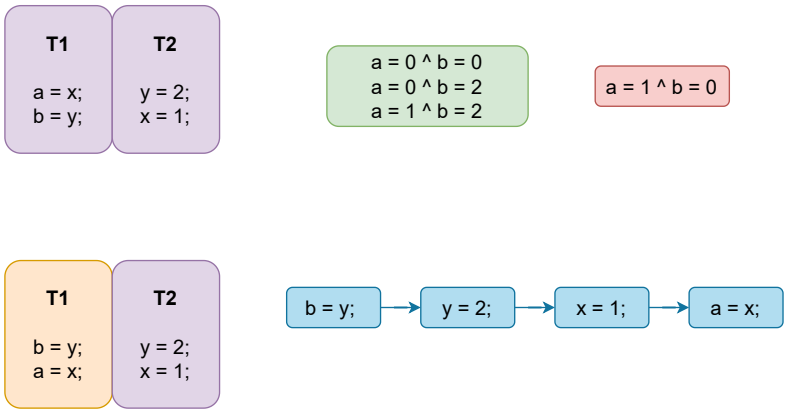


Figure 2.2: The original program above with its allowed and disallowed outcomes in SC, followed by the program with the two reads reordered below, justifying the disallowed outcome under SC.

The reordered program can justify a sequential interleaving of events to have the

disallowed outcome in the original program.

Such concerns are not only related to simple reordering, but also something such as elimination. Yet another example can be that of elimination. Consider the program in Figure **??** below

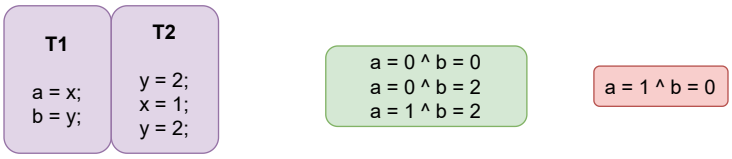


Figure 2.3: Another program above with its allowed and disallowed outcomes under SC.

In this example, the red box outcome is still not allowed under SC. We can notice though that the write to y in $T2$ is done twice. Naturally, the compiler might think of eliminating one of them under the context of redundant code elimination. Suppose it eliminates the first write $y = 2$. Then the resulting program as shown below in Figure 2.4, can justify the outcome under SC whcih was disallowed in the original program.

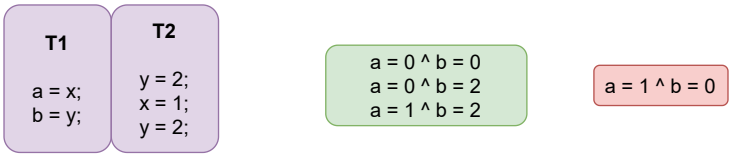


Figure 2.4: Program after elimination of $y = 2$, justifying disallowed outcome under SC.

The above examples show that even simple transformations can be unsound under SC. Complex program transformations such as register allocation, common-subexpression elimination, loop invariant code motion are some examples which use the above two basic transformations heavily. Having them unsound under SC also implies the compiler is not allowed to do a varying class of optimization without breaking the consistency rules udner which the concurrent program is supposed to behave / execute. In addition, hardware at that time had come up with several features such as buffers and cache systems, which in principle could be used effectively for performance, but were not so useful for programs respective SC semantics. In order to effectively critique a model in terms of program transformations, it is quite

impractical to come up with exhaustive examples of program to justify their validity. A proof in this sense would be required.

Ševčík et al. [23] showed that standard compiler optimizations were rendered invalid under the respective memory model of Java. Simple transformations such read-after-wrtie elimination or redundant read introduction which play a role in major performance based transformations such as common sub expression eliminations were unsound. Morisset et.al [24] showed the soundness of optimizations with non-atomic memory accesses in the C11 memory model. (ALSO FILL THE REST HERE) Vafeiadis et al. [25] showed that common compiler optimizations (including those with atomic memory accesses) under C11 memory model were invalid, followed by proposing some changes to allow them. Transformations such as sequentialization, strengthening access modes, and even roach-motel reorderings were unsound. Their proposed changes to the model have been incorporated by the standard committe for C11.

With respect to instruction reordering and redundancy elimination in shared memory programs, Ševčík et al. [26] gave a proof design on how to show such optimizations are valid. This approach relies on the idea of reconstructing the original execution of a program given the optimized one, while also showing the well known SC-DRF guarantee holds—programs that are *data-race-free* (DRF) must exhibit SC semantics. Our approach is in fact the other way round; we show that the optimized program does not introduce new behaviours, by explicitly using the consistency rules to show that relevant ordering relations are preserved. We do not show it specifically only for *data-race-free* programs as the model that we refer to also requires that programs with race have a defined behavior.

2.3 Axiomatic Style Specifications of Weak Memory

While most programming language semantics are defined and analyzed operationally, memory consistency models have been shown to be more effective for analysis using an axiomatic specifcation. This axiomatic specification typically is in terms of defining partial orders between relaxed memory events (read/write) and then specifying restrictions on the composition of these partial orders. A very elaborate literature on axiomatic semantics of weak memory is given by Alglave et al. [27]. This work introduces a new tool called *herd*, which can be used for testing program examples against memory models. The memory models themselves have to be specified in axiomatic format.

Typically, such an axiomatic approach to something related to concurrency avoids dealing with the state space explosion of operational models, which is often quite difficult to reason with given so many relaxed memory based outcomes. We can completely avoid the problems of reasoning with seemingly infinite states of programs by reasoning with partial orders between events in a concurrent program. We in our approach also rely on this axiomatic perspective, using which we prove the validity of two powerful program transformations used in most compiler optimizations.

In our literature review, such an axiomatic perspective traces back to times when problems of relaxed memory accesses were being addressed only at the hardware level. Sindhu et al. [28] specifies an axiomatic framework for specifying the behaviors of shared memory multiprocessors. Owens et al. [8], Batty et al.[16] specify the respective memory models in an axiomatic format.

2.4 Other Concerns

Upto date, memory consistency models are still lacking a very user-friendly specification for use by many programmers. In addition to this, we also showed that related work existed exposing limitations in many models with respect to basic program transformations.

Compilation There is also the big question of compilation; whether the compilation from source and target with different memory models is correct? This question is also quite open ended. While we did not do literature reivew in this direction particularly, works done by Lahav et.al [18] and Watt et.al **Watt** on the C11 and JavaScript memory model respectively exposed incorrect compilation done due to which the program exhibited unintended behaviors on execution. The interested reader can refer to (INSERT IMM LINK HERE FROM VIKTOR'S SITE) to explore further in this direction.

Verificaiton / Model Checking While the focus of this thesis is not on Formal Verification, there certainly does exist several research works in verifying weak memory programs. One of the well known methods for verifying concurrent programs is using *Stateless Model Checking*. There are two works in this explored during literature review that are exemplary of the use of Stateless Model Checking in weak memory programs. Michalis et.al [29] provided a model checker named RCMC for performing effective model checking of relaxed memory programs in C11. Michalis

et.al [30] provided a model checker named GenMC for performing effective model checking parametric to relaxed memory models.

Out of Thin Air Many memory models face the concern which is notoriously known as *out-of-thin-air*; a program can give an outcome that is not supposed to have existed in any observable behaviors as per the semantics. Such behaviors have been shown to be in programs that exhibit *data-races* in their executions. The C11 memory model [31] escapes from addressing it by stating that any program with *data-races* has undefined behavior. Java on the other hand, relies on a complicated semantics to guarantee that no *out-of-thin-air* values can exist in programs with *data-races*. This to date is still very complicated and subtle to understand in order to use it properly in programs. While this may seem that *out-of-thin-air* is a bad property that must be avoided at all costs, Verbrugge et.al [32] showed that disallowing them also disallows quite a few compiler optimizations.

Custom Memory Models We also explored certain works that came up with their own memory model in order to simplify the problems above that may arise due to it. Arvind et.al [33] presented a novel framework to specify memory models using Instruction Reordering and Store atomicity. This work in our eyes, was a better representation of intended behaviors that should be allowed by a memory model. Marino et.al [34] proposed a new memory model named *DRFx*, which they claimed to be quite intuitive for programmers as well as allowing many of the program transformations responsible for major performance benefits. Kang et.al [35] proposed a new memory model referred to as *Promising Semantics* to address the well known out-of-thin-air problem that exists in current memory models. Though the semantics of this model, in our experience is quite complex and not so user-friendly.

Our analysis is based on this corrected model by Watt et al. [20] which is incorporated in the ECMAScript draft specification. As far as our knowledge goes, no analysis has been done on this model to identify its implications on standard compiler optimizations.

As a summary, this chapter elicited the key researhc works done in the domain of relaxed memory models, its specificaiton and its impact of program transformations. In the next chapter, we state the problems in the existing specification of the ECMAScript memory model, followed by a more formal specificaiton of the same.

Chapter 3

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 3.1: The definition for Coherent Reads

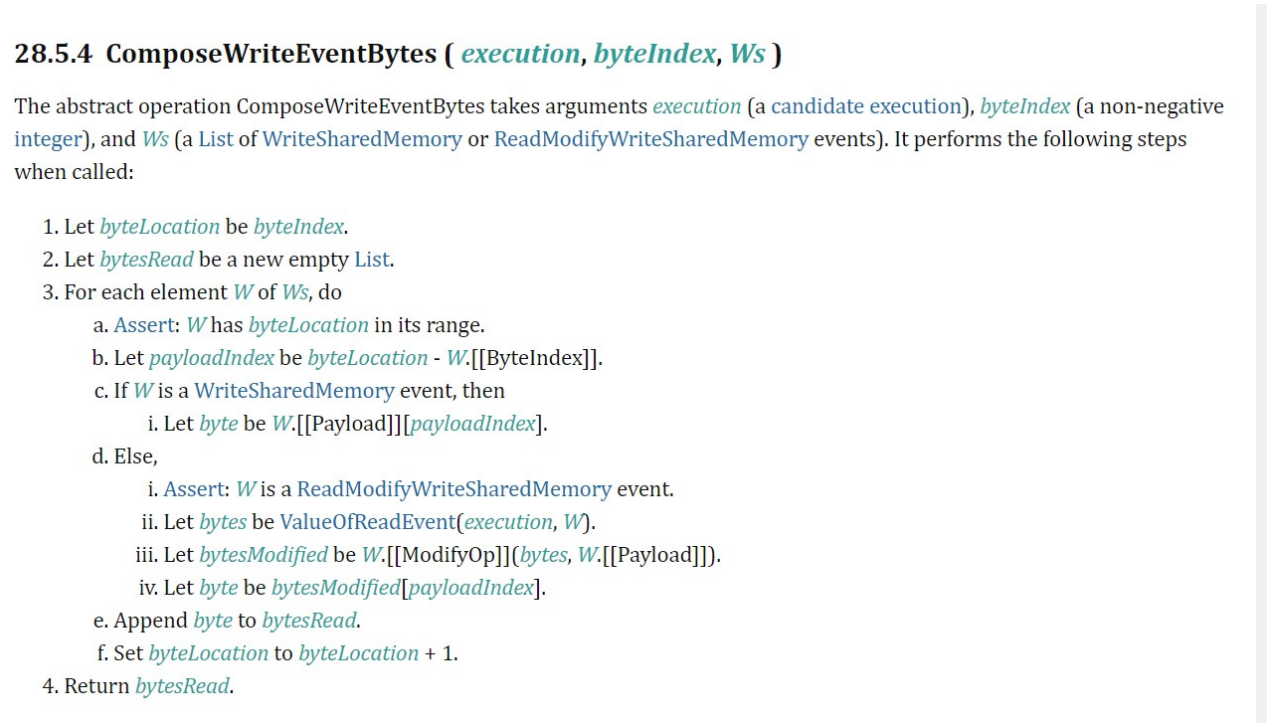
The above axiom is specified as a return value to an abstract operation. While understanding this requires one to know thedefintions of Ws, $execution$, $SharedDataBlockEventSet$ abstract operation, etc. we believe this is not needed as to understandwhat the axiom is about, which informally can be stated 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 3.2: 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 3.3: 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.

3.1 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.

3.2 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.

3.2.1 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].

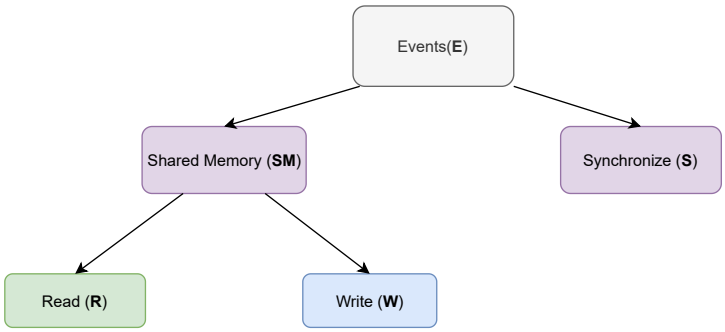


Figure 3.4: The hierarchical categories of different sets of events.

3.2.2 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 $\Re(e) = \Re(d)$

3.2.3 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: $event_{type}$.

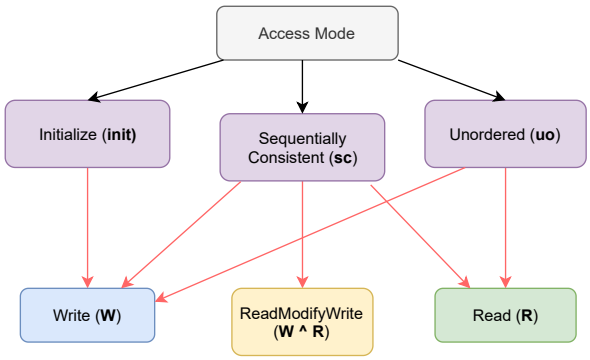


Figure 3.5: Access Modes

3.2.4 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: $event_{tf/!tf}$.

3.3 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.

3.3.1 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.

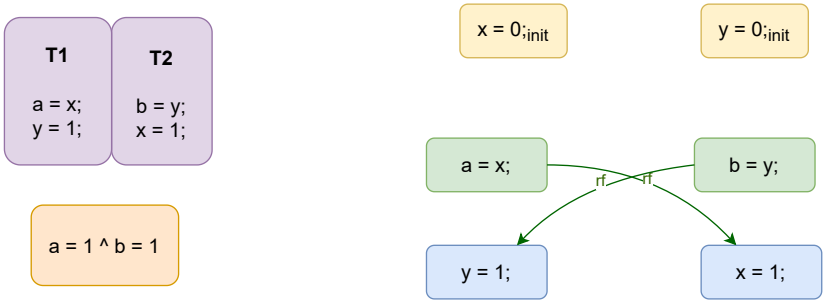


Figure 3.6: An example to show the reads-from relations that are drawn for the example program between read and write events.

3.3.2 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.

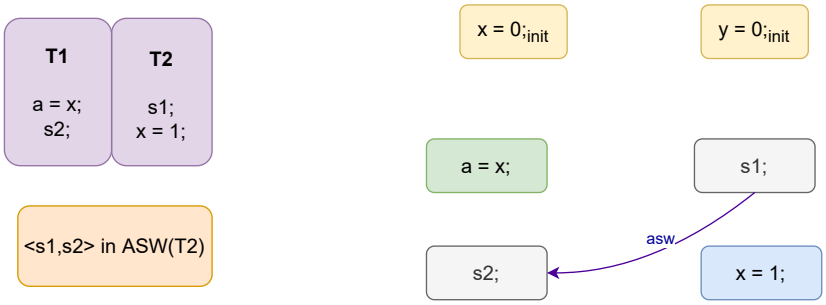


Figure 3.7: An example to show the reads-from relations that are drawn for the example program between read and write events.

3.4 Ordering Relations among Events

3.4.1 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 $d \overrightarrow{ao} e$.

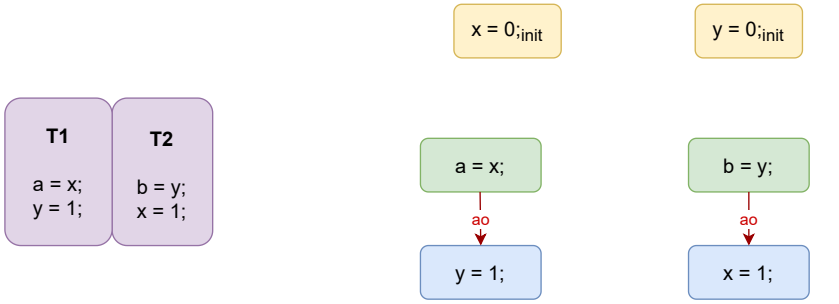


Figure 3.8: An example with agent order among the events.

[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.

3.4.2 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].

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

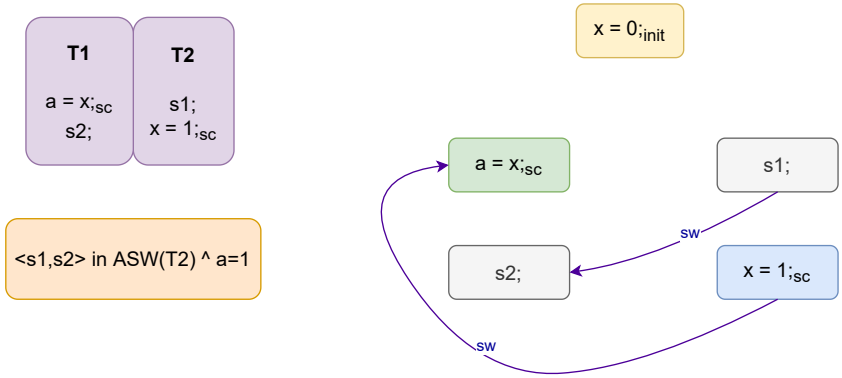


Figure 3.9: An example with synchronize with relations among the events.

3.4.3 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.

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

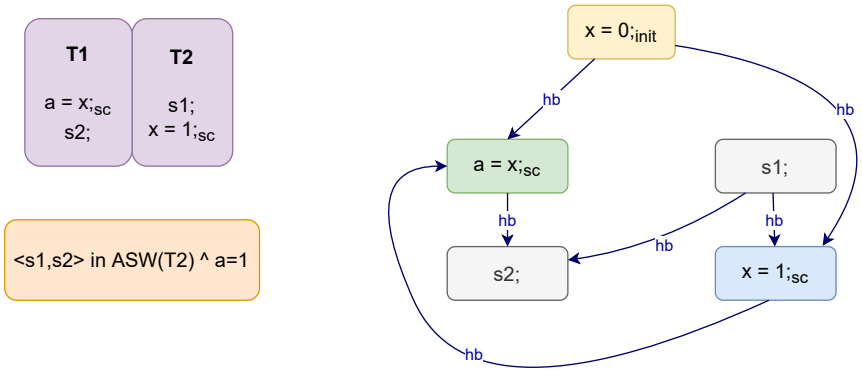


Figure 3.10: An example with all the types of happens-before relations between events.

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

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

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

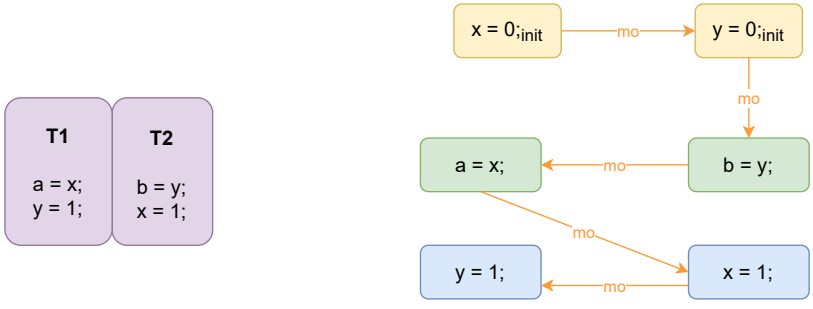


Figure 3.11: An example with a memory order (total) among all events.

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.

3.5 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 3.5.1. *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 3.5.2. *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 3.12.*

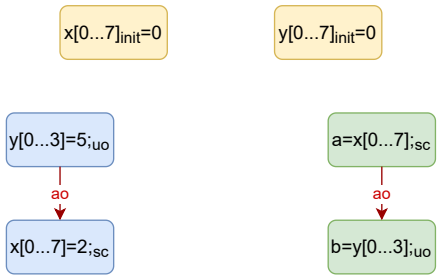


Figure 3.12: An example of a Candidate

Definition 3.5.3. *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.*

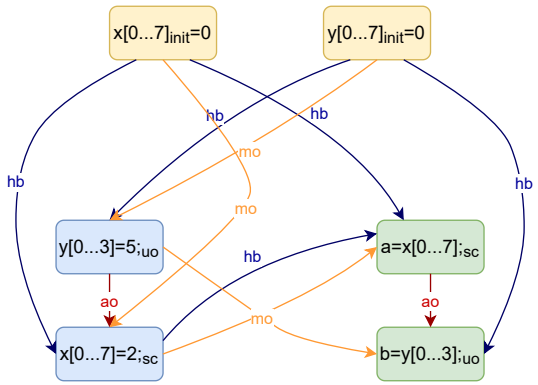


Figure 3.13: An example of an Execution based on Candidate above

Definition 3.5.4. *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.

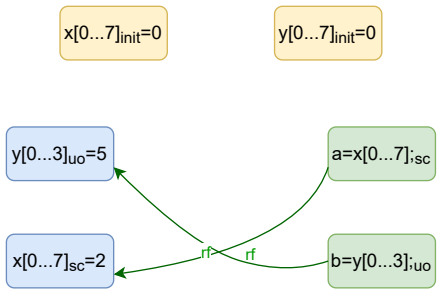


Figure 3.14: Observable Behavior

Definition 3.5.5. *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.

3.6 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*.*

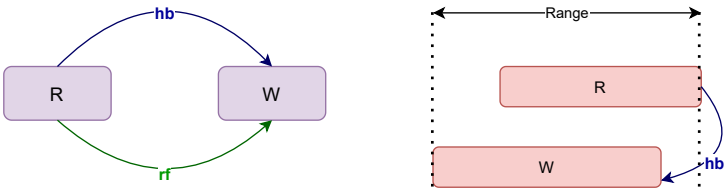


Figure 3.15: A read value cannot come from a write that has happened after it

- *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*.*

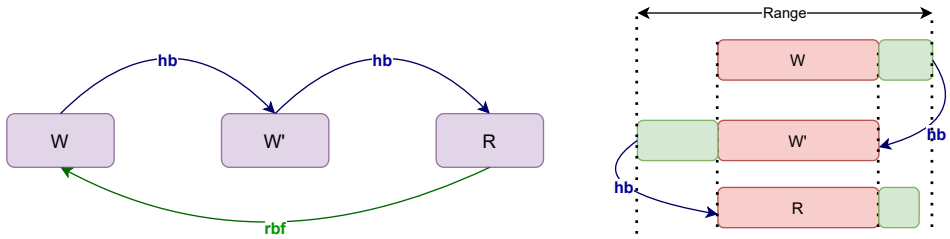


Figure 3.16: A read value cannot come from a write if there is a write that happens between them, writing to the same memory:

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.

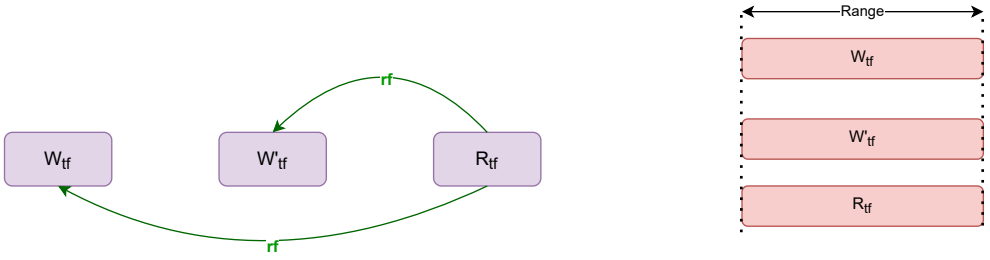


Figure 3.17: Pattern of Tear-free reads

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.

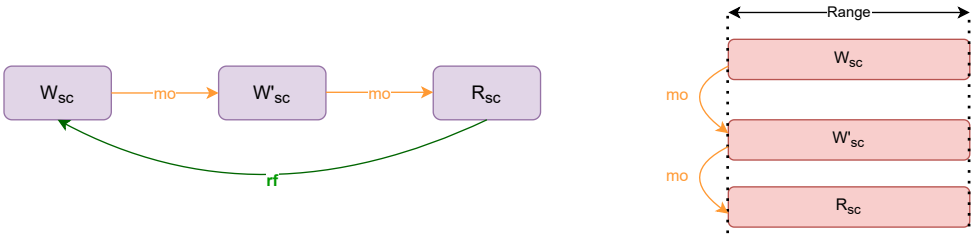


Figure 3.18: A read value cannot come from a write, if there exists a write memory ordered between them and all 3 events are sequentially consistent with equal ranges.

- *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.

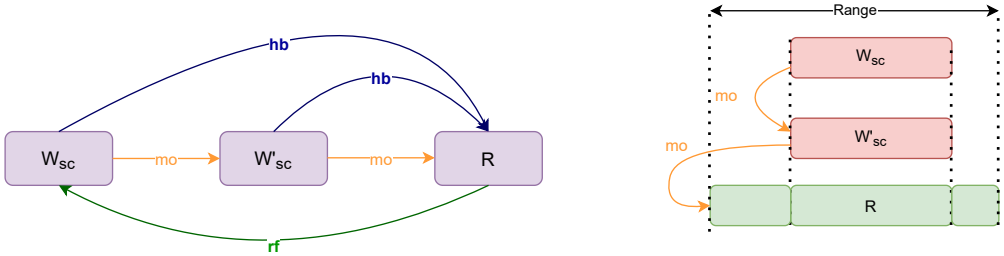


Figure 3.19: A read value cannot come from a write, if there exists a write memory ordered between them and both writes are sequentially consistent with equal ranges.

- *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.

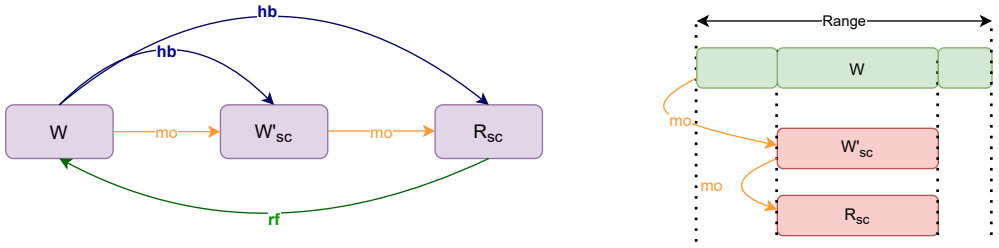


Figure 3.20: A read value cannot come from a write, if there exists a write memory ordered between them and both this write and the read are sequentially consistent with equal ranges.

3.7 Race

3.7.1 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,

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

3.7.2 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).*

3.8 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 4

Instruction Reordering

This chapter addresses the validity of instruction reordering under the ECMAScript memory model. We first start by showing some examples of Candidate Executions where reordering is not safe in the relaxed memory context. We give a brief summary of our approach towards a proof to identify when such a reordering is safe. Next, we introduce few more defintions for our purpose followed by two basic lemmas that will be instrumental for proofs in this chapter and the next. We then formulate a theorem and a corresponding corollary that covers validity of reorderinng at a Candidate Execution level. Lastly, we address reordering at the program level (still abstracted to a set of shared memory events) involving loops and conditional branching. 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.

4.1 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:

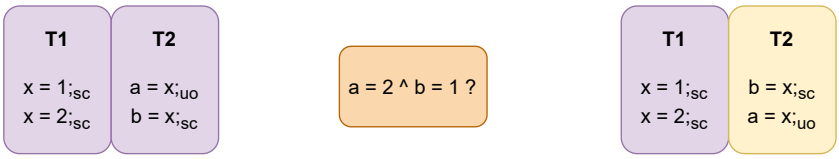


Figure 4.1: Ex1(a)

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.

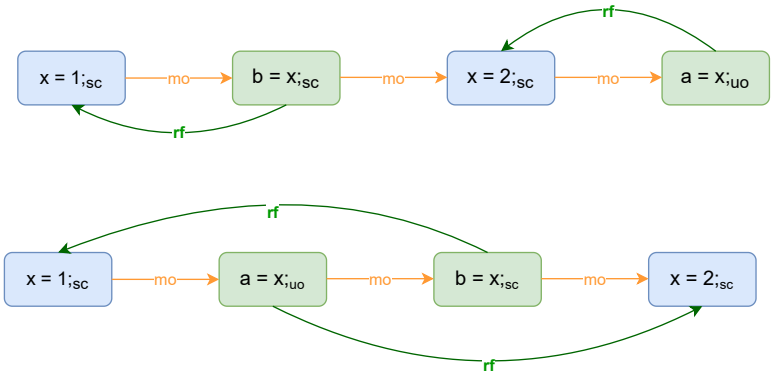


Figure 4.2: Ex1(b)

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:

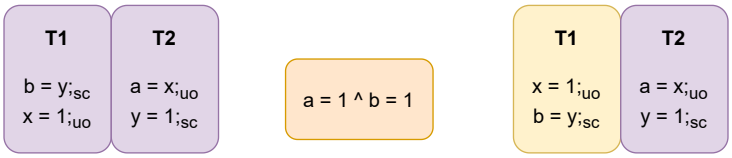


Figure 4.3: Ex2(a)

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.

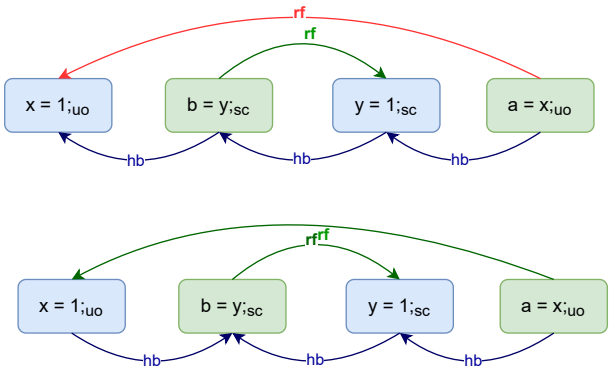


Figure 4.4: Ex2(b)

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.

4.2 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, become infeasible as the programs scale in length and complexity. This is because of the exponential increase in possible outcomes as the number of threads and program size increases. Hence, generalizations by using a small examples 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.

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

$$e \overrightarrow{_{ao}} d \longmapsto d \overrightarrow{_{ao}} e$$

What implications this change has on the Observable Behaviors 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 $\overrightarrow{_{hb}}$ relations (other than the one we eliminate intentionally i.e $e \overrightarrow{_{hb}} d$) would suffice for our purpose. Since $\overrightarrow{_{mo}}$ respects $\overrightarrow{_{hb}}$, we in turn even preserve the memory order which is essential.

4.3 Some Useful Definitions

Before we go about proving when reordering is valid, we would like to have two additional definitions which would prove useful[1].

Definition 4.3.1. *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* $\overrightarrow{ao}$ *relation among them and are* next to each other*, which can be defined formally as*

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

Definition 4.3.2. *Direct happens-before relation (dir) We define* dir *to take an ordered pair of events* (e, d) *such that* $e \overrightarrow{hb} d$ *and gives a boolean value to indicate whether this relation is direct, i.e those relations that are not derived through transitive property of* $\overrightarrow{hb}$*.*

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 4.3.3. *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.*

[1]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.

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

4.4 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,

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

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 4.3.2 and Def 4.3.1, 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

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

In the second case, we have

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

Thus, from Def 4.3.2 and Def 4.3.1, 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 4.3.2 these are only relations satisfying $dir(d,k)$. Thus, we can once again infer that for any $k \neq e$

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

The following figure summarizes the intuition behind both cases:

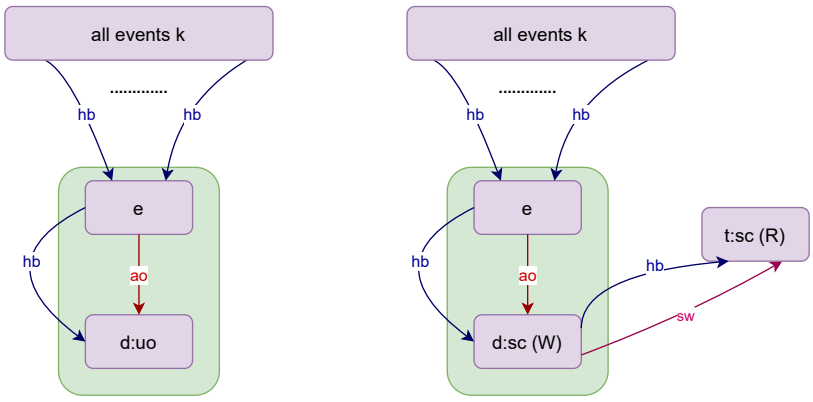


Figure 4.5: Left figure is for first case and right one for the second

□

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,

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

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 4.3.2 and Def 4.3.1, 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 $\xrightarrow[ao]{}$ is a total order, d would be the only such event. Thus, for any other event $k \neq d$, we can infer,

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

In the second case, we have

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

Thus, from Def 4.3.2 and Def 4.3.1, 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 4.3.2 these are only relations satisfying $dir(k, e)$. Thus, we can infer that for any $k \neq d$

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

The following figure summarizes the intuition behind both cases:

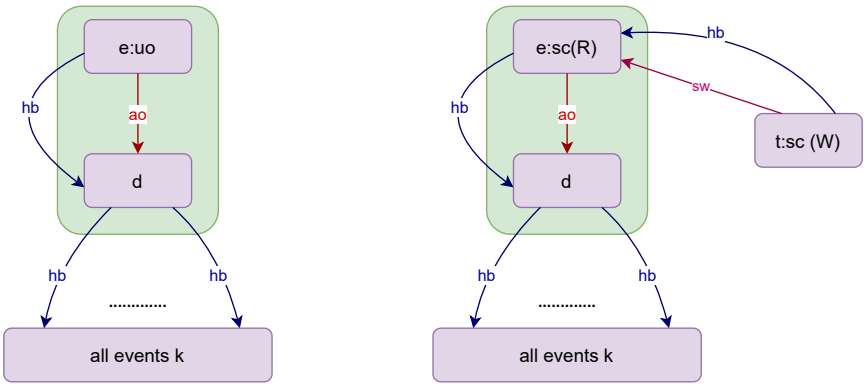


Figure 4.6: Left figure is for first case and right one for the second

□

4.5 Valid reordering at the Candidate Level

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.

4.5.1 Reordering of Consecutive Events

Theorem 4.1. *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*

$$cons((,e),d) \ \wedge \ e \ \overrightarrow{ao} \ d.$$

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, they 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. 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.

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

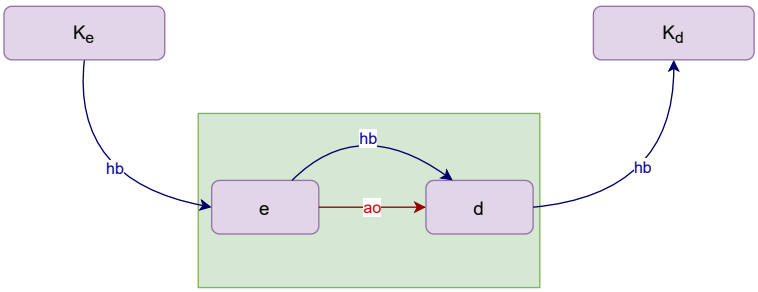


Figure 4.7: For any Candidate Execution of C, the set K_e and K_d

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

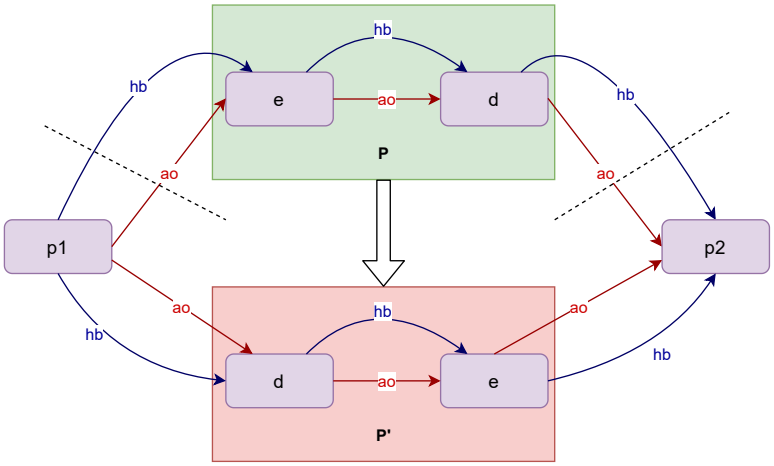


Figure 4.8: The direct relation changes that can be observed while reordering events e and d

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

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

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[2]$<$ $p1, p2 >$

<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 4.9: 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, $K_e = K_{e1} + K_{e2} + p1 + p_x$

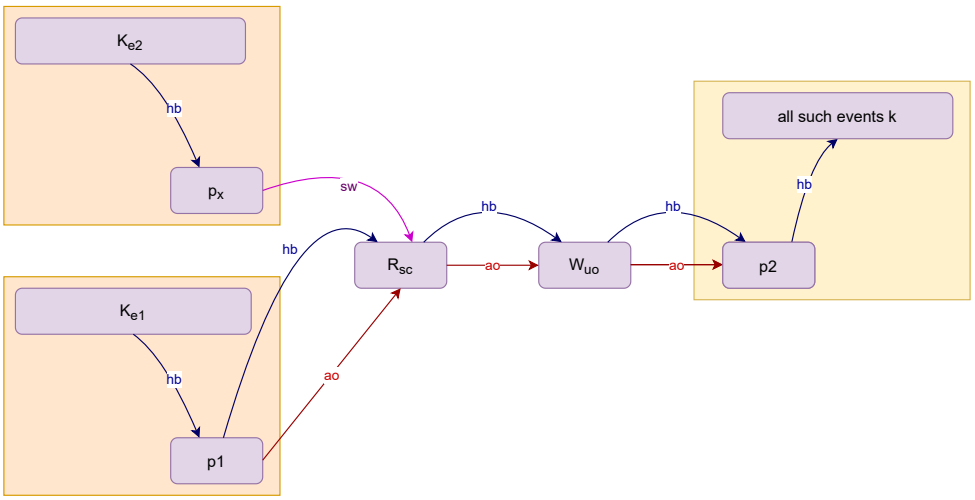


Figure 4.10: A Candidate Execution where p1 is not a valid pivot

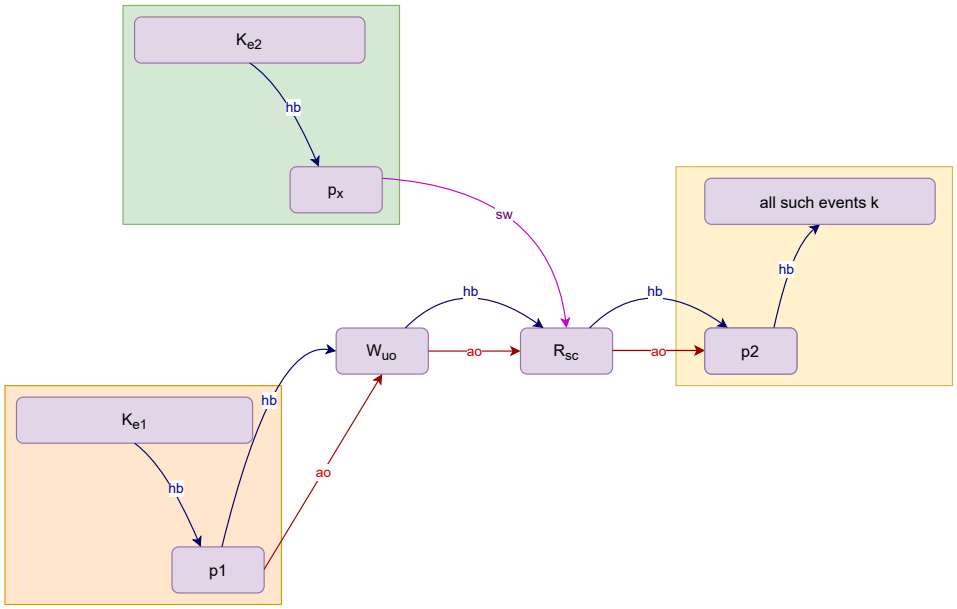


Figure 4.11: The resultant Candidate Execution after reordering, exposing the relations with p_x, K_{e2} and d that arelost

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

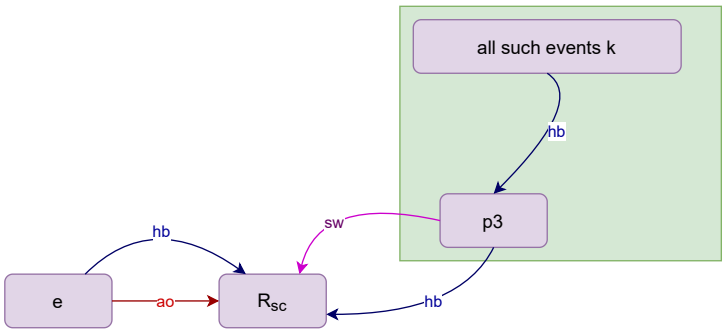


Figure 4.12: A Candidate Execution where d is a sequentially consistent read

[2]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.

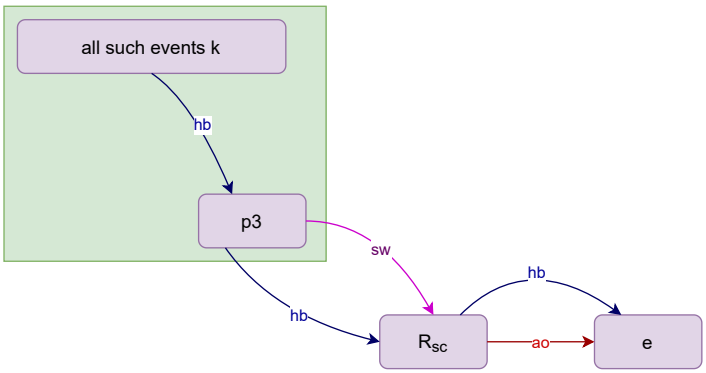


Figure 4.13: The Candidate Execution after reordering, exposing the new relations established with e, $p3$ and set k

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 4.14: 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. This is because the model requires that $\overrightarrow{hb}$ is a strict partial order.

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[3].

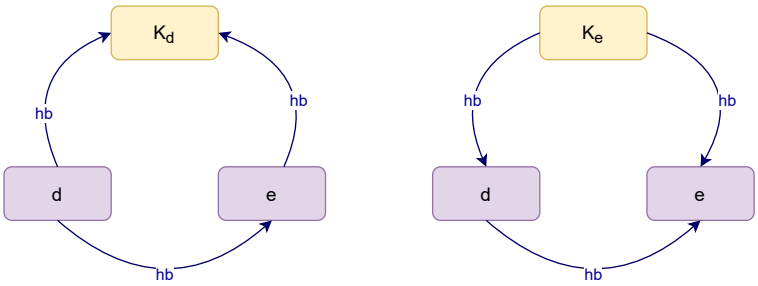


Figure 4.15: If k belongs to one of the sets K_e or K_d

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

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

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

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

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

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

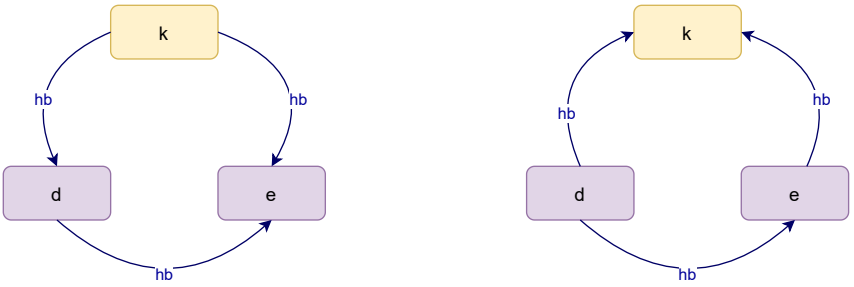


Figure 4.16: If $k \xrightarrow[hb]{} e$ or $d \xrightarrow[hb]{} k$ are new sets of relations

[3] 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.

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, $e \xrightarrow[hb]{} k$[4]. Thus, we have a cycle. The following figure shows this pictorially

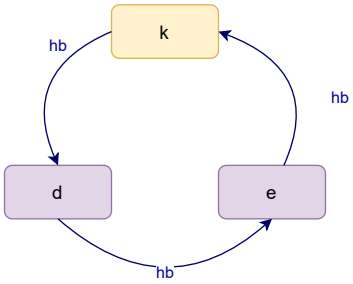


Figure 4.17: A cycle exists in the case where we have two new sets of relations ($k \xrightarrow[hb]{} e$ and $d \xrightarrow[hb]{} k$)

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

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

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

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

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

So, the following is also a cycle

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

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

[4]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.

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	N	N	N	N
uo-sc	Y	N	Y	N
sc-uo	N	N	Y	Y
sc-sc	N	N	Y	N

Table 4.1: Insert good caption here.

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 $d \xrightarrow[hb]{} e$. This new relation itself could introduce new observable behaviors. Hence, 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 inferring 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.

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

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

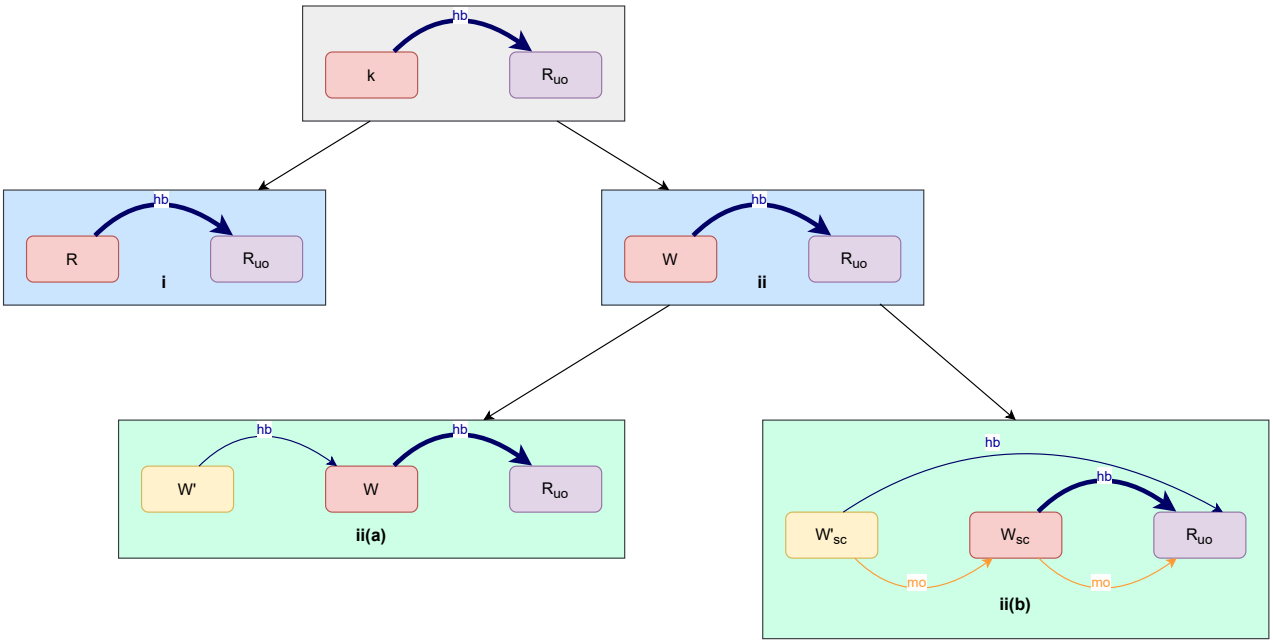


Figure 4.18: The role of the axioms on introducing a new relation between an unordered Read and some event k

- 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.

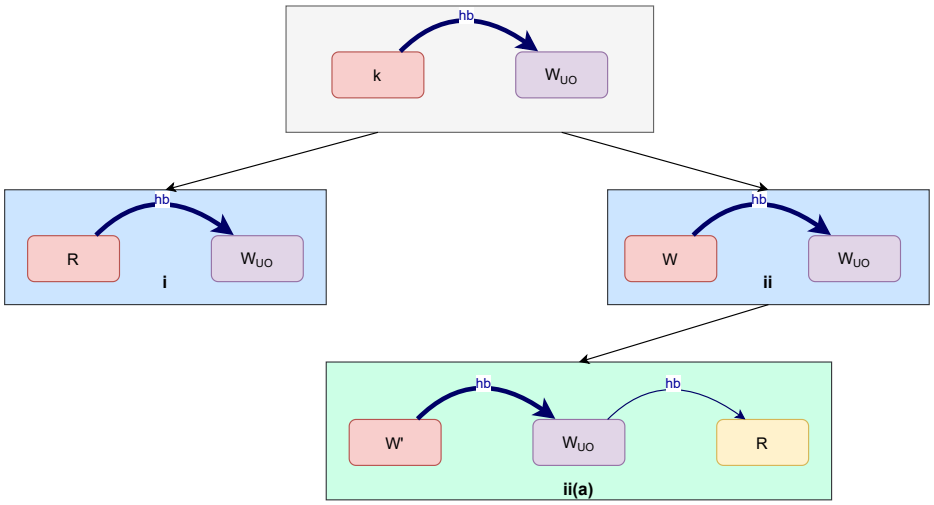


Figure 4.19: (i) and (ii(b)) satisfy the axiom of Coherent Reads

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.

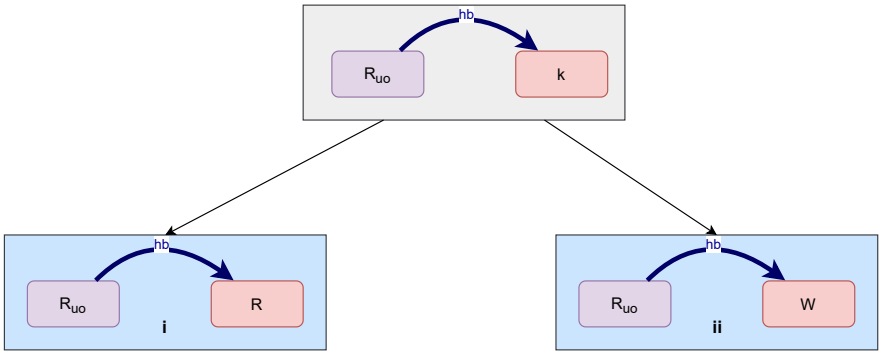


Figure 4.20: (ii) satisfies the axiom of Coherent Reads

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.

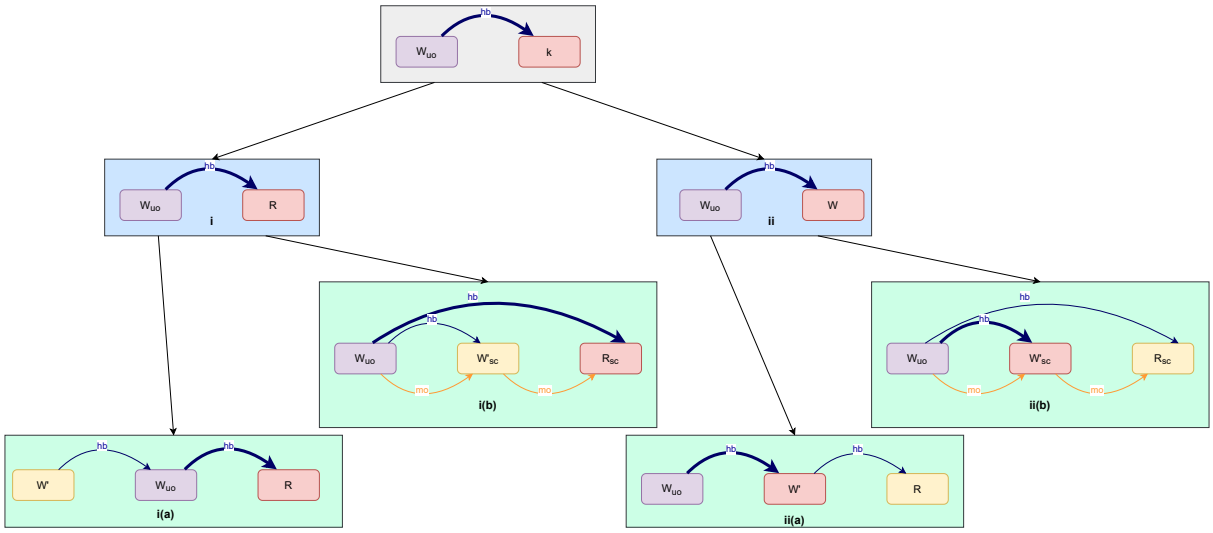


Figure 4.21: (i(a)), (ii(a)) satisfy the axiom of Coherent Reads, whereas (i(b)), (ii(b)) satisfy the axiom of SequentiallyConsistent Atomics

For case (d) we can observe the following

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

The table above, precisely is the definition of a reorderable pair (after including the constraints on ranges). If we write the above table in the form of an expression we have an expanded format of our Reorderable pair function.

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

□ □

4.5.2 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 4.1.1. *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

$$e \overrightarrow{ao} k \overrightarrow{ao} d \wedge\ cons(e, k)\ \wedge\ cons(k, d).$$

What we want after reordering is C', with

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

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 4.1, reorder them, giving us candidate C'' with

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

whose observable behaviors is subset of C.

Similarly, now with $Reord(e, d)$, by Theorem 4.1, we get candidate C''' with

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

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 4.1, giving us our final candidate C' with

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

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 4.1, 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 4.1, 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 4.1.2. *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 4.1.3. *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.

4.5.3 Counter Examples for all the Invalid Cases

For cases where reordering is not safe to do, we also show counter examples of programs where new observable behaviors are introduced. This additionally will help gain intuition about the proof given. Note that we do not show examples for cases where $d \xrightarrow[hb]{} e$ itself is sufficient to show a new observable behavior, as this is a trivial exercise that can be done just using sequenital programs. We show counterexamples where $\xrightarrow[hb]{}$ relations lost (those accross agents specifically), could introduce new observable behaviors.

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 illustrates when reordering two reads to x as per the specification of their access orders and range results in an observable behavior disallowed.

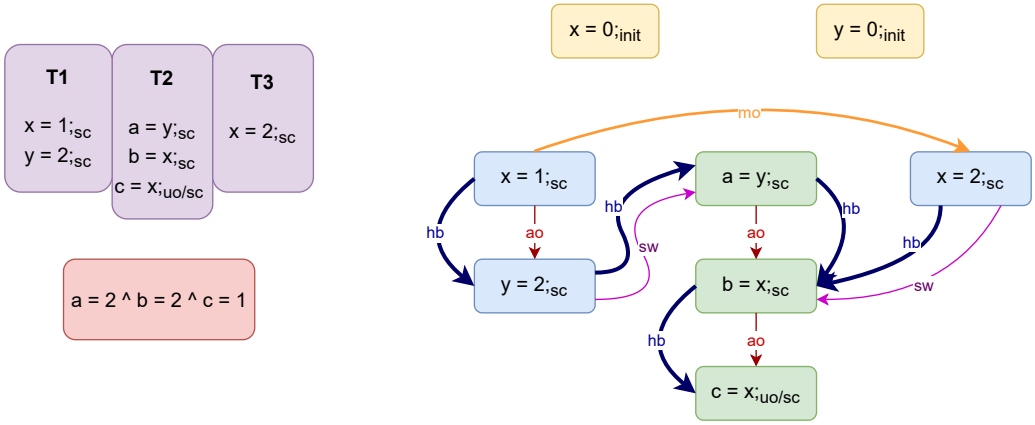


Figure 4.23: Case where a = 2 , b = 2, c = 1 is invalid due to Sequentially Consistent Atomics

The figure on the left above shows an example of a candidate where the case of outcome 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 $x = 1;_{sc} \xrightarrow{hb} b = x;_{sc}$.
- Because $\{x = 2;_{sc}\} \xrightarrow{sw} \{b = x;_{sc}\}$, this means the read value of b is 2.
- From the above two, we can infer $\{x = 2;_{sc}\} \xrightarrow{mo} \{x = 2;_{sc}\}$.
- We can then also infer that $\{x = 1;_{sc}\} \xrightarrow{hb} \{c = x;_{uo/sc}\}$ and $\{x = 2;_{sc}\} \xrightarrow{hb} \{b = x;_{uo/sc}\}$
- By Axiom 3 pattern 1 and 3, the read value for c cannot be 1.

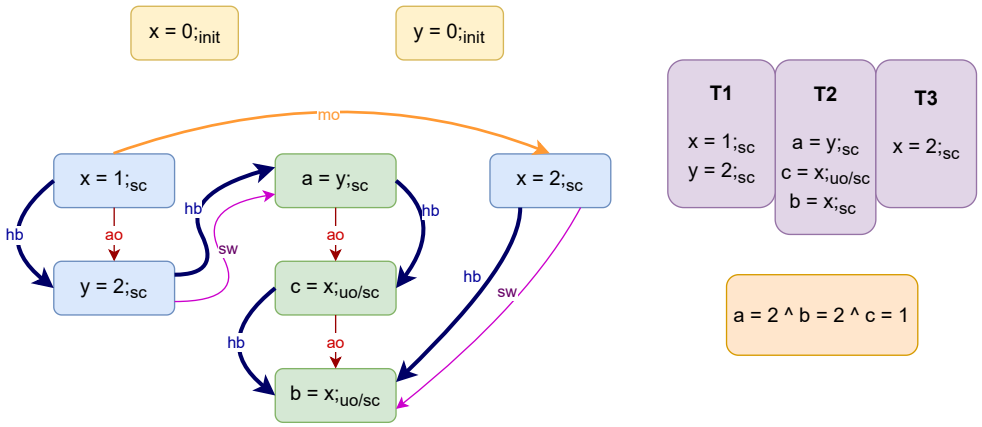


Figure 4.24: Case where the reads are reordered and a = 2 , b = 2, c = 1 is valid

The figure on the right shows the program after reordering the two reads in $T2$, 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

- We can infer from the Candidate Execution that $\{x = 1;_{sc}\} \overrightarrow{hb} \{c = x;_{uo/sc}\}$.
- No Axiom has restrictions on $\overrightarrow{rf}$ between the above two events.
- Hence, the read value of c can be 1.
- Further, the memory order is not inferred yet[5], hence, the read value for b can be 2.
- Hence the reordering of the two reads is invalid.

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.

[5]Note that if the memory order was reversed in the original candidate execution, Axiom 3 would restrict the value of b to be 1. Since this is not possible due to the synchronized relation established, it must be the case that $x = 1$ is memory ordered before $x = 2$.

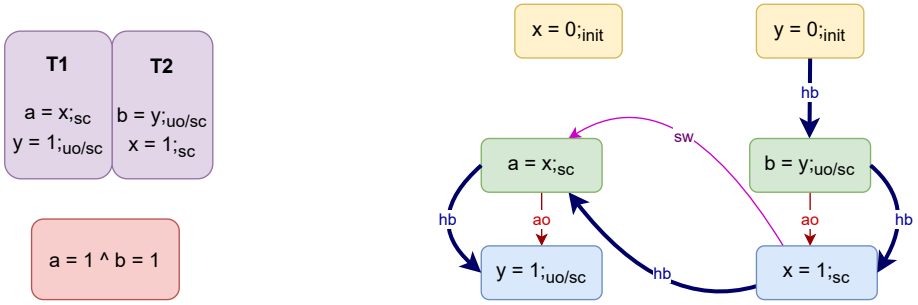


Figure 4.25: Case where a = 1 and b = 1 is invalid due to Coherent Reads.

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

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

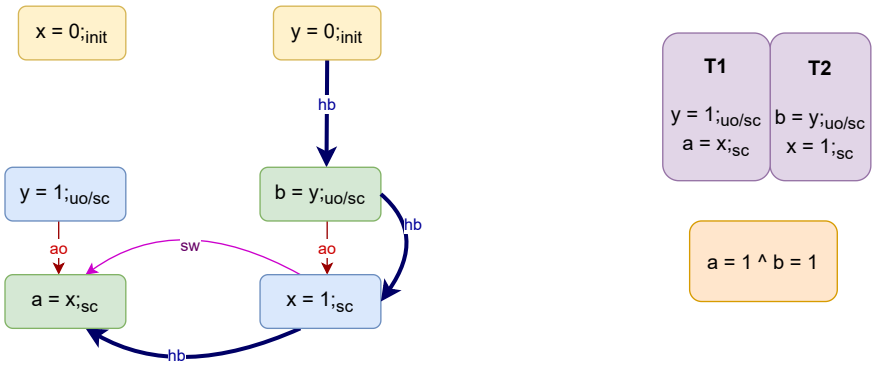


Figure 4.26: Case where events of T1 are reordered, resulting in a = 1 and b = 1 to be valid.

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

- From the Candidate Execution, we can infer $\neg\{b = y_{uo/sc}\} \overrightarrow{hb} \{y = 1_{uo/sc}\}$

- Since there is no $\overrightarrow{hb}$ 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.

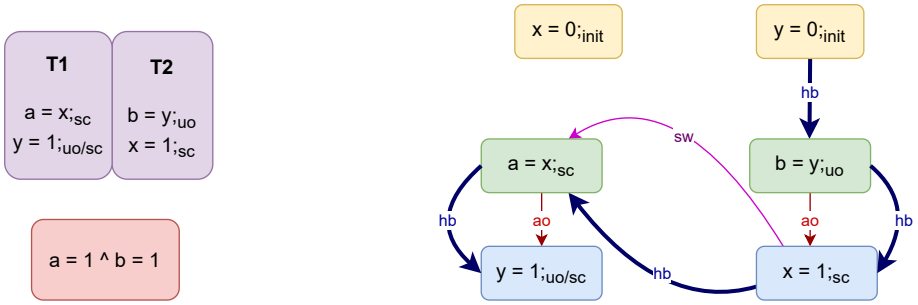


Figure 4.27: Case where a = 1 and b = 1 is invalid due to Coherent Reads.

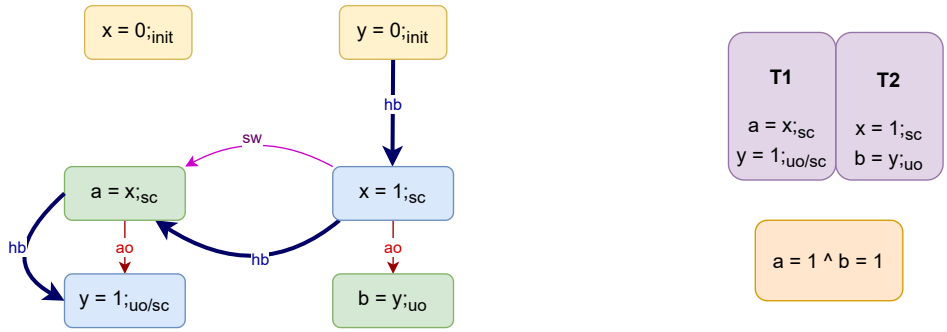


Figure 4.28: Case where events of T2 are reordered, resulting in a = 1 and b = 1 to be valid.

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

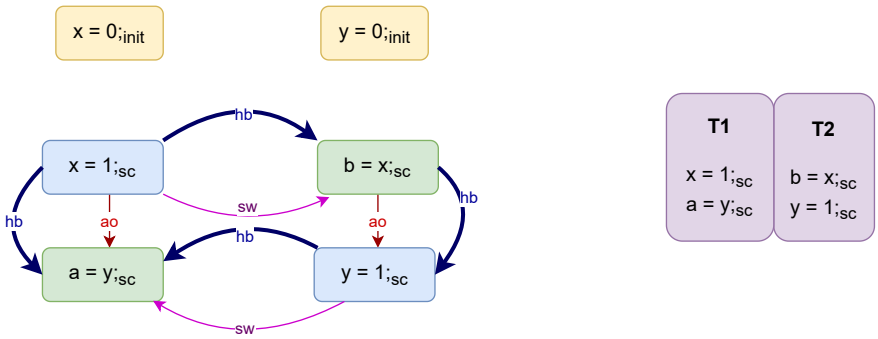


Figure 4.29: Case where a = 1 and b = 1 is valid and no happens-before cycles

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

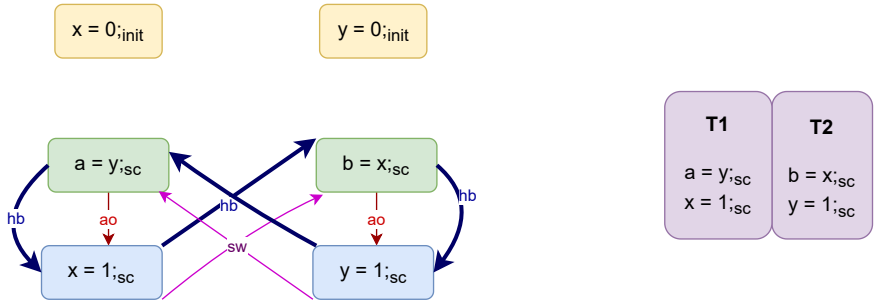


Figure 4.30: Case where a = 1 and b = 1 is creates a happens-before cycle

Observation:

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

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

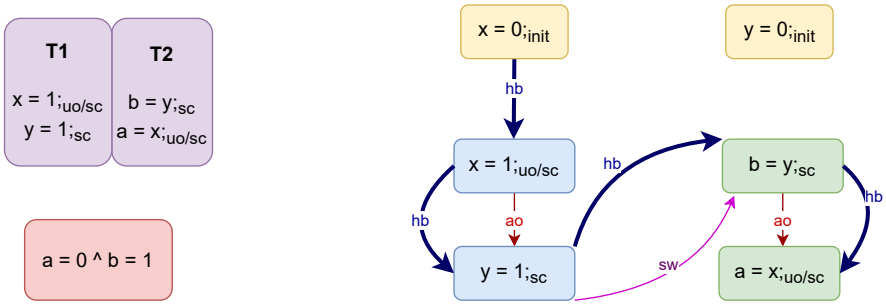


Figure 4.31: Case where a = 0 and b = 1 is invalid due to Coherent Reads.

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

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

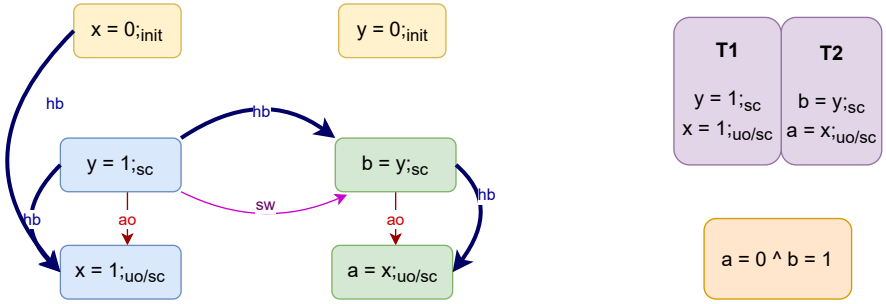


Figure 4.32: Case where events of T1 are reordered, resulting in a = 0 and b = 1 to be valid.

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

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

4.6 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.

4.6.1 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[6]*.*

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.

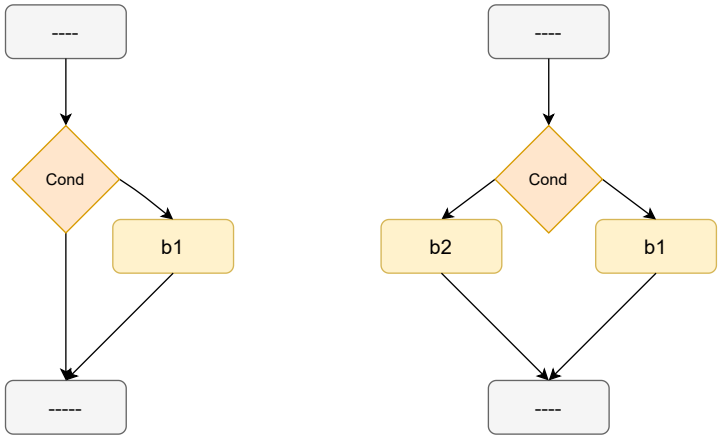


Figure 4.33: Two forms of conditonals

We use the above two properties to state the following lemma

Lemma 3. *Reordering an event* e *outside a conditional violates Property 1 and Property 2*

Proof. Let $B1$ represent the set of events in a conditonal branch in program P such that $e \in B1$.

Let us consider the first property. By Property 1, we have

$$\exists C \in P \text{ s.t. } \forall b \in B1, \ b \notin C$$

[6]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.

On reordering event e outside a conditional branch, we have

$$\exists C \in P \text{ s.t. } \forall b \neq e \in B1,\ b \notin C$$

thus violating the Property 1 that must hold for the original program P.

Now let us consider the second property. Let $B2$ represent the set of events in the alternative branch to $B1$. Then by Property 2, we have

$$\nexists C \in P \text{ s.t } \forall b \in B2,\ e \in C \wedge b \in C$$

On reordering event e outside the conditional branch, we have

$$\exists C \in P \text{s.t } \forall b \in B2,\ e \in C \wedge b \in C$$

thus violating Property 2, that must hold for the original program P.

□

The above lemma's proof also lets us infer that on reordering an event outside a conditional, there are *new* Candidates exist with a new event belonging to it. We use this insight to state the following corollary for reordering under conditionals.

Corollary 4.1.4. *Consider a program P with conditional branches and its candidates $C_1, C_2, ..., C_n$ in which events e and d 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 and its corresponding program P'. If the following two conditions hold:*

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

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

Proof. We prove the second condition first. Assume the second condition does not hold. Then we would have

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

By Property 1, e or d must belong to a conditional branch.

If e and d are in different branches of same conditonal, then by Prop 2 there wouldn't exist any candidate C in P where we could reorder e and d. If e and d are of the same conditonal branch, and neither one of them belong in any conditional branch nested within, then our above assumption does not hold. (simple sequential property of conditional branches)

For the other cases, wihtout loss of generality, let us suppose the first condition holds, i.e.

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

The cases for the above can be summarized in the figure below:

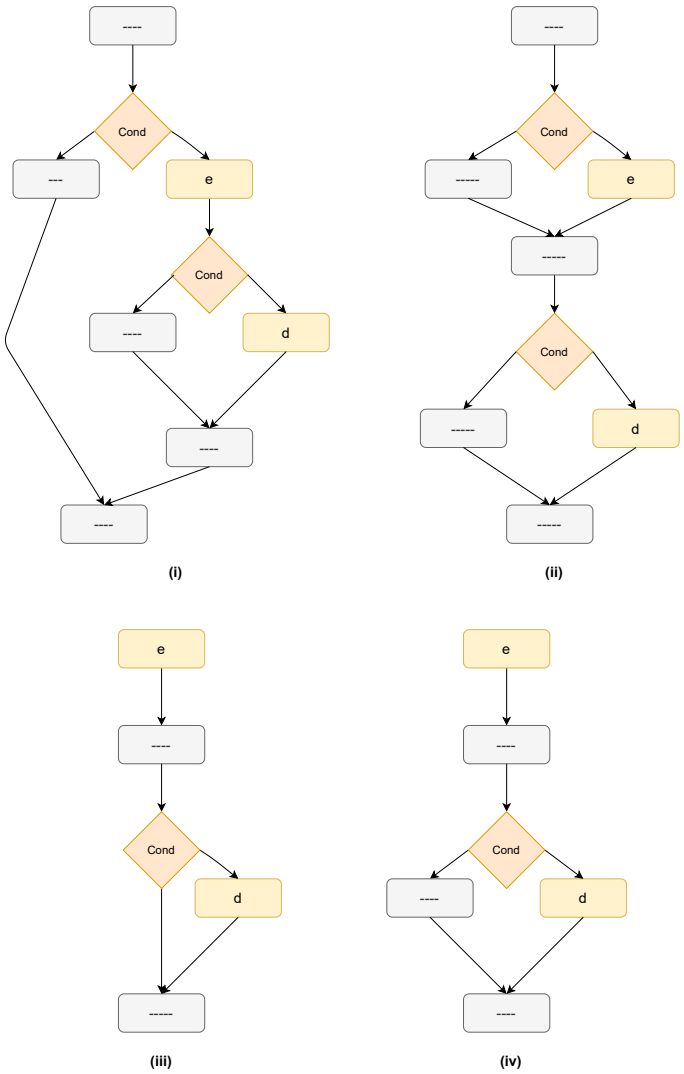


Figure 4.34: Four cases where e or d could be part of some conditional branch.

For cases (i) and (ii), by Lemma 3, a new Candidate with event e or d exists without their respective conditional branches being taken. For cases (iii) and (iv),

by Lemma 3, a new Candidate with event d exists without its respective conditional branch being taken. Irrespective of e or d being a read or a write, there could be a new $\overrightarrow{rf}$ relation be formed with some event k in the Candidate. By Def **??**, we have a new observable behavior[7].

Hence, by contradiction, the second condition must hold.

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 4.1.1, thus giving us that the observable behaviors of C_i' is a subset of C_i. By property of unions of sets, we can infer that the set of Observable Behaviors of P' is a subset of that of P.

□

4.6.2 Counter Examples for Programs with Conditionals

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.

The example below shows a case with a conditonal having two branches where reordering e and d is not safe.

[7]Note that this argument is purely in terms of the execution graphs. The new event can possibly have a new reads-from relation established with some event in the graph itself. Since this new node did not exist in the graph before, and since every node in the graph is considered unique, we can infer that a new observable behavior is introduced. Analyzing which such execution graphs are equivalent, would imply drawing equivalence between two differnt reads-from relations. This could be done as a whole by addressing redundancy introduction optimization. This is not within the scope of the thesis.

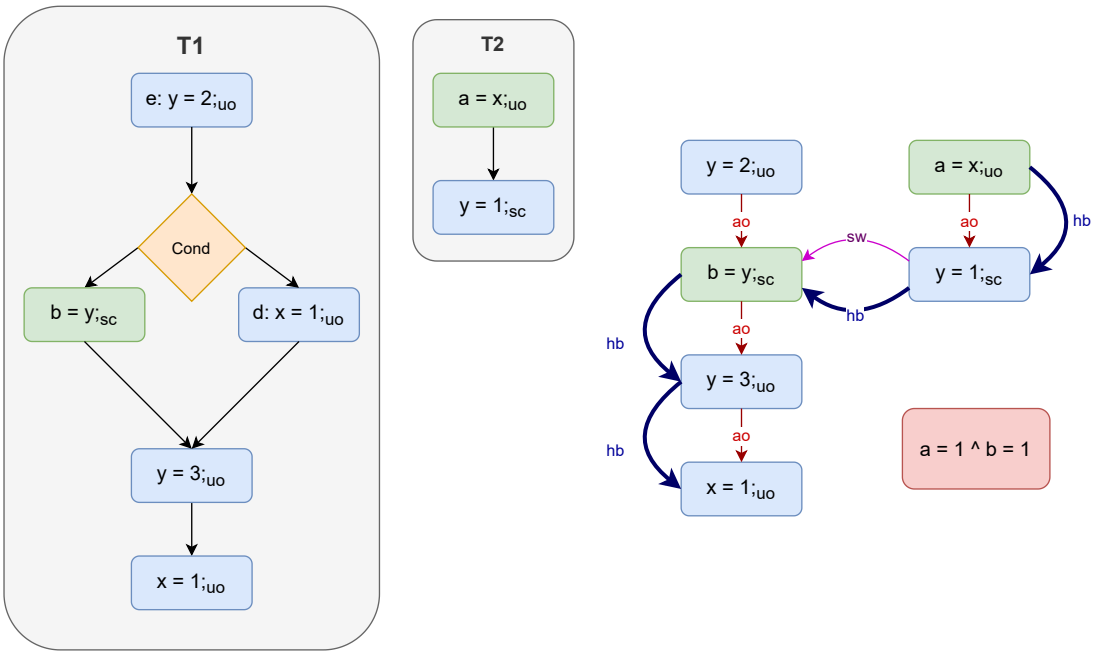


Figure 4.35:

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

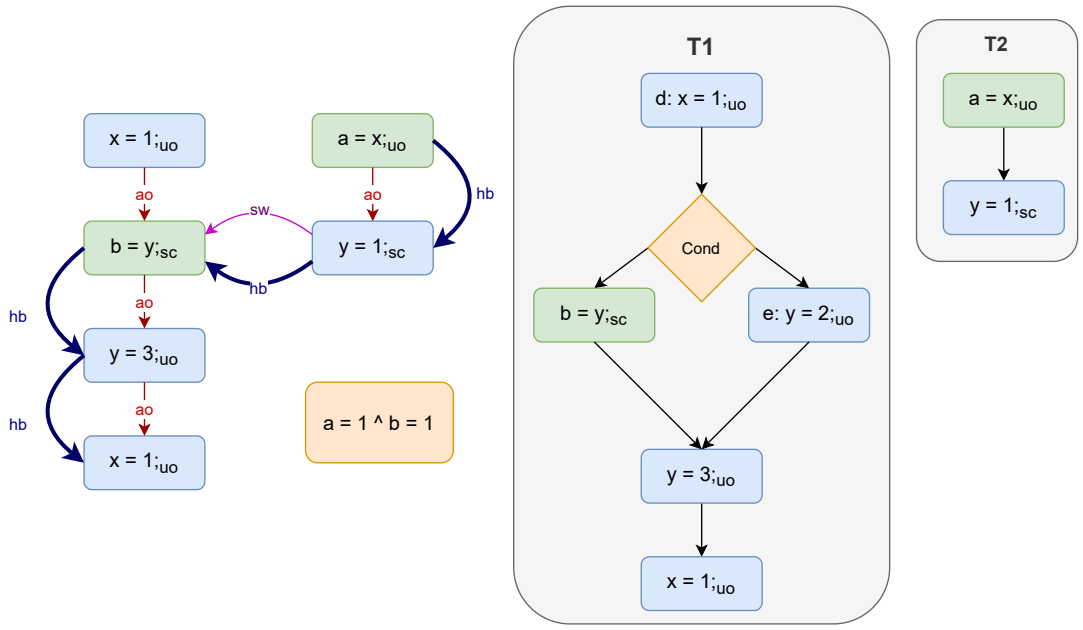


Figure 4.36:

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[8].

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

The example below is a counter example where the conditional has only one branch.

[8]Notice that the above counterexample can also be attributed to introducing a new write $x = 1$ in a candidate.

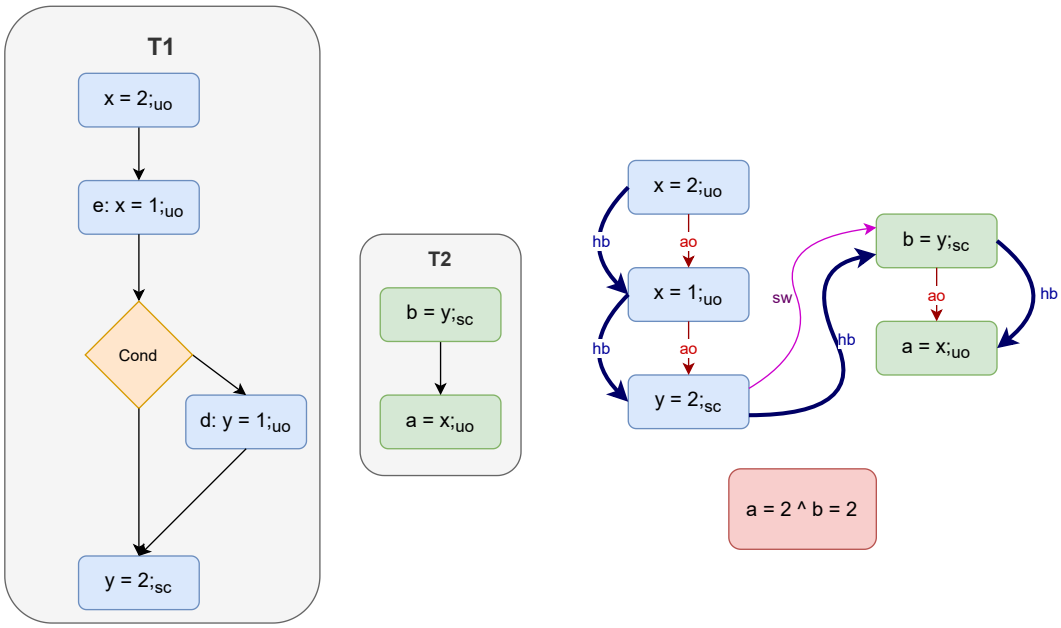


Figure 4.37:

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

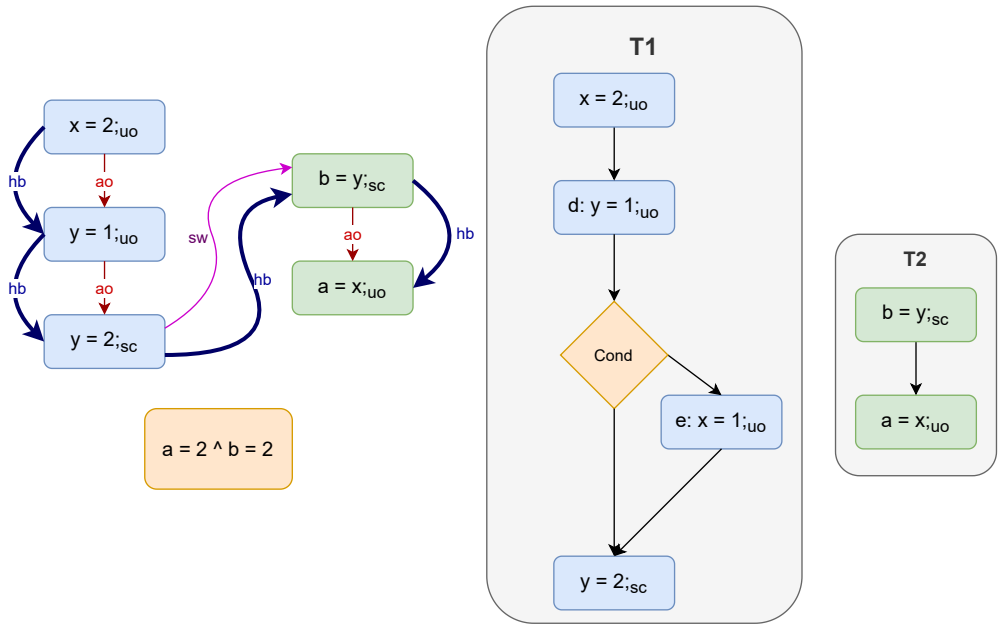


Figure 4.38:

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[9].

- 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.

We leave the rest of the cases as an exercise to the avid reader[10].

[9] Notice that the above counterexample can also be attributed to the elimination of a write $x = 1$.

[10] 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.

4.6.3 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 define the following corollary for reordering events e and d within a loop. The intuition is that if we can reorder e and d in every iteration of the loop, then the observable behaviors of the resultant program is a subset of the original.

Corollary 4.1.5. *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:*

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

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 4.1. Condition 2 and 3 correspond to Corollary 4.1.1 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[11].

□

[11]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. The reason we did not define it this fashion is because this would need a formal definition of "between". But 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.

Reordering Accross Loops 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.

The problem is that we cannot use parent Candidate to generate resultatnt Candidate of the transformed program. Reordering at the Candidate level does not map directly to Reordering that is done to perform something like Loop invariant code motion.

We will show in the next chapter that we can infact define one of its forms, viz. 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 5

Elimination

This chapter addresses the validity of elimination under the ECMAScript memory model. We first start by showing some examples of Candidate Executions where write elimination is not safe in the relaxed memory context. We then give an explaination of why read elimination may not be safe to do. We then formulate two theorems, one for read elmination and one for write with a corresponding corollary for write elimination. Similar to reordering, we address elimination at the program level (still abstracted to a set of shared memory events) involving loops and conditional branching. We lastly formulate theorems and their corresponding proof which shows how loop invariant code motion can be described as a combination of elimination and reordering at the Candidate execution level. We conclude by mentioning those cases of code motion that cannot be explained yet using our two program transformations.

5.1 Elimination

Many programs which contain loops or are representing big softwares fall victim to having many redundant code. This could also be possible due to differnt phases of optimization the compiler performs, which leaves certain residual code in each phase. One such example of redundant code is that when there are two consecutive reads/writes to the same memory location. Another example could be the case that after many optimization passes, certain memory values are not used by the resultant

program itself, so the compiler may decide to remove it.

Consider to give concrete optimization examples here from your notes on optimizations (Clark's lectures).

In a sequential setting the effect of removing such code can be addressed relatively easily as sequential code does not have the complexity of multiple outcomes. However, as we saw for instruction reordering, in a concurrent setting, elimination may not be that straightforward.

Let us, for instance, consider a program where we have two writes to the same memory writing the same value and the program after eliminating the latter write. Let us then consider another program with the latter write eliminated.

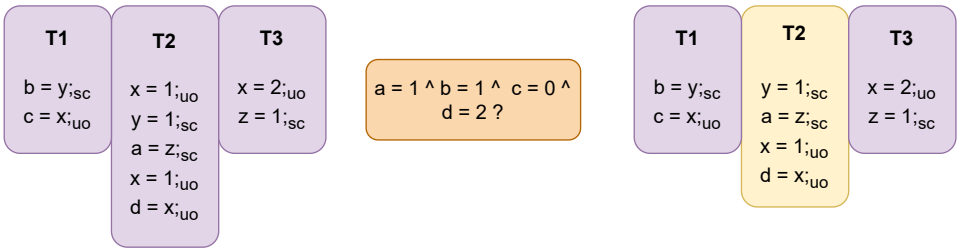


Figure 5.1: Ex1(a)

The orange box shows the observable behavior that we want to consider. In the first program, such an outcome should not be allowed. While in the program after eliminating the latter write, this outcome is allowed. The following figure explains the relations formed in a candidate execution that disallow the behavior in question for the original program but justify it after elimination.

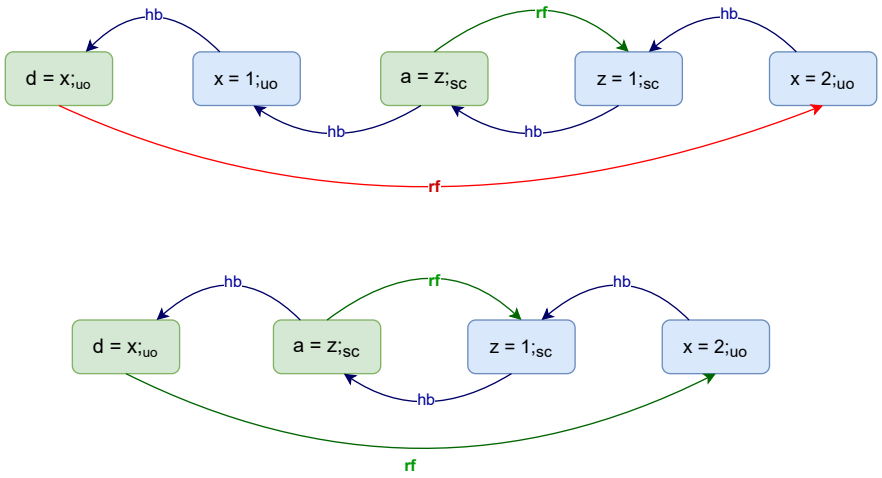


Figure 5.2: Ex1(b)

Have a better caption for the above figures later. TO better explain them.

The first set of relations is for the original program, where Axiom 1 prohibits the read d to have value of x as 2. The second set is for the modified program, where none of the axioms restrict such a behavior (I would suggest the avid readers to check it for themselves).

An example where reads is unsafe to be eliminated is not quite easy to construct. This is because the semantics of the model does not really have any read-read dependance; one read value does not affect any subsequent(agent ordered) read value to same memory unless constrained by happens-before. So one might assume that we can freely eliminate reads: this however would not be safe to do, due to other reasons such as forward progress. If for instance, a loop repeatedly checks the value of some memory (say x) and will terminate only if the memory reads some fixed value, each candidate execution representing each iteration of the loop would represent also the minimum iterations before the read is of fixed value. And this value would be of the read in the last iteration.

For addressing the validity of eliminations under our memory model, we separately address first Read elimination, followed by Write elimination, both at the candidate level. We finally address them at the program level with conditionals and loops involved.

Theorem 5.1. *Consider a candidate C of a program and its possible Candidate Executions where $\overrightarrow{hb}$ is strictly partial order. Consider an event e in C such that $e \in R$. Consider another Candidate C' without the event e. If e has an unordered*

access mode, then the set of Observable behaviors of C' is a subset of C without the relation $e \xrightarrow[rf]{} w$ where w is some write event in C.

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

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

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

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

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

We need to ensure the following relations hold after elimination.

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

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

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

By Lemma 1, $e : uo$ is the only case where p_b can be a valid pivot. By Lemma 2, $e : uo \ \vee \ e : sc$ are the cases where p_a can be a valid pivot. We need both the above conditions to be satisfied to have a valid pivot pair. Hence, $e : uo$ is the only possibility in which a valid pivot pair can exist.

Put a figure here to show this pivot role.

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

$$k \xrightarrow[hb]{} e \ \vee \ e \xrightarrow[hb]{} k \tag{5.1}$$

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.

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.

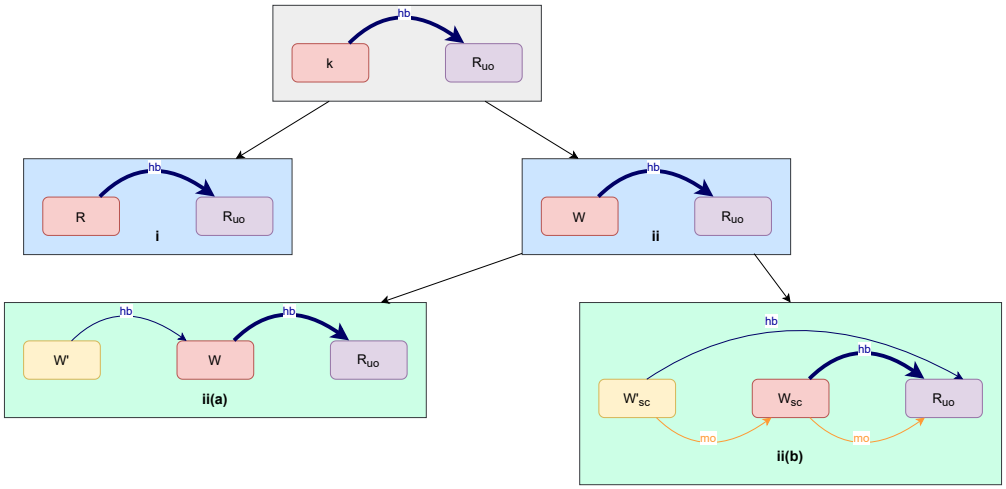


Figure 5.3: The first type of relations removed and the various patterns forbidden by them.

Observations:

- (i) is not a pattern forbidden by the consistency rules.
- (ii)(a) is a pattern of Axiom 1, however, only restricting $\overrightarrow{rf}$ relation with R and W'(which here is our Unordered Read)
- (ii)(b) is a pattern of Axiom 3, however, once again, only restricting $\overrightarrow{rf}$ relation with R and W'.

In the second case, we have the following possibilites.

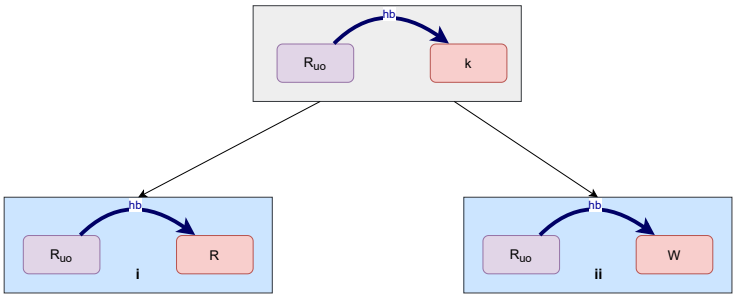


Figure 5.4: The second type of relations removed and the various patterns forbidden by them.

Observations:

- (i) is not a pattern in any Consistency rules
- (ii) is a pattern of Axiom 1, however, only restricting $\overrightarrow{rf}$ relation with R and W

From the above observations, we can infer that the relations removed only have restriction on reads-from relations on the event e we eliminate. Thus, 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 5.2. *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 in C such that*

$$cons(e, d) \ \wedge \ e \overrightarrow{ao} d.$$

Consider a Candidate C' without event e. If e has an unordered access mode and e and d have the same range,

$$e : uo \ \wedge \ \Re(e) = \Re(d)$$

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

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

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

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

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

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

We need to ensure the following relations hold after elimination.

$$\forall k_a \in K_a \ \wedge \ \forall k_b \in K_b \ . \ k_b \xrightarrow[hb]{} k_a \tag{5.2}$$

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

$$\forall k_b \neq p_b \in K_b \ . \ k_b \xrightarrow[hb]{} p_b \tag{5.3}$$
$$\forall k_a \neq p_a \in K_a \ . \ p_a \xrightarrow[hb]{} k_a \tag{5.4}$$

By Lemma 1, $e : uo \vee e : sc$ are the cases where p_b can be a valid pivot. By Lemma 2, $e : uo$ is the case when p_a (which in our case here is d) can be a valid pivot. Since we need a valid pivot pair, $e : uo$ is the only case where this is possible.

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

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

$$k \xrightarrow[hb]{} e \ \vee \ e \xrightarrow[hb]{} k \tag{5.5}$$

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.

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:

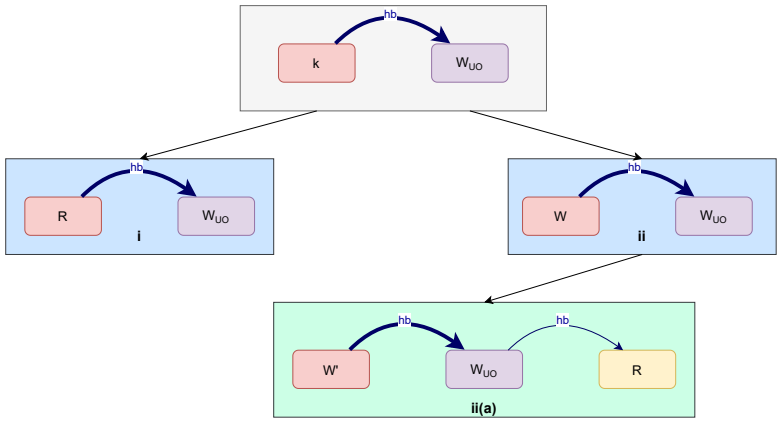


Figure 5.5: First case possibilities (change caption stimiar to that for read elim)

We can observe the following:

- (i) is a pattern from Axiom 1 that restricts the read R reading from e. This will remain the case even after elimination of e.
- (ii)(a) is a pattern from Axiom 1, forbidding R to read some bytes of W'. This will remain the case after elimination of e if firstly we have $d \xrightarrow[hb]{} R$ and $W \xrightarrow[hb]{} d$. By Lemma 2, the first relation holds and by Def of happens-before the second holds. Secondly, we need to ensure that after elimination, Axiom 1 now restricts the exact set of $\xrightarrow[rbf]{}$ relations with W' and R as before. Since R or W' can be arbitary, we would need the ranges of e and d to be same.

For the first case, we have the following possibilities:

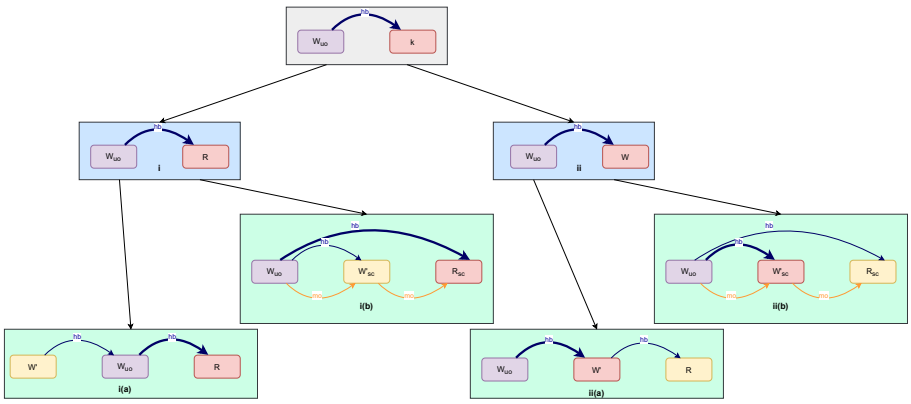


Figure 5.6: Second case possibilities (change caption stimiar to that for read elim)

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 from Axiom 3, which restricts R from reading anything of W. This will reamin the case after W is eliminated.
- (ii)(a) is a pattern from Axiom 1, 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.

This can be slightly modified to show that the range of e can even just be contained in d. This would ensure we can eliminate writes in more other cases.

□

Corollary 5.2.1. *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:

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

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

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

whose observable behaviors are a subset of C.

By Def 4.3.1 and Theorem 5.2, we can eliminate e, thus giving us candidate C' with

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

whose observable behaviors are a subset of C''.

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

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

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

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

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

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

By Theorem 4.3.3 and Def 4.3.1 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}.

□

5.2 From Candidates to Program

5.2.1 Addressing Programs with Conditionals

Elimination under conditionals can be tricky. For instance, while performing write elimination at the candidate level, we require a write to be agent ordered ahead of it with no intervening read in between (aas stated by the Theorem 5.2 and Corollary 5.2.1). In the case of conditionals, the resultant candidates may or may not have such a write depending on whether the conditional is satisfied. While performing read elimination, the read itself can be the conditional check, which then brings the question of why this read would be eliminated by the compiler. Note again that we do not assume anything about the optimization being done by the compiler. We only investigate whether eliminating an event is safe to do.

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

Corollary 5.2.2. *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 the following two conditions hold

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

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

Proof. 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 5.2.1, 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.

Suppose the second condition does not hold, then we have

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

This would mean, for such a candidate and its Candidate Executions, by Corollary 5.2.1 the observable behaviors of C' may not be a subset of C. Hence observable behaviors of P' may not be a subset of P. Hence, by contradiction, the second condition must hold.

Note that the second condition, by Prop 1 implies e and d cannot be part of different conditional branches.

We elicit the cases of conditionals that are allowed by the second condition below.

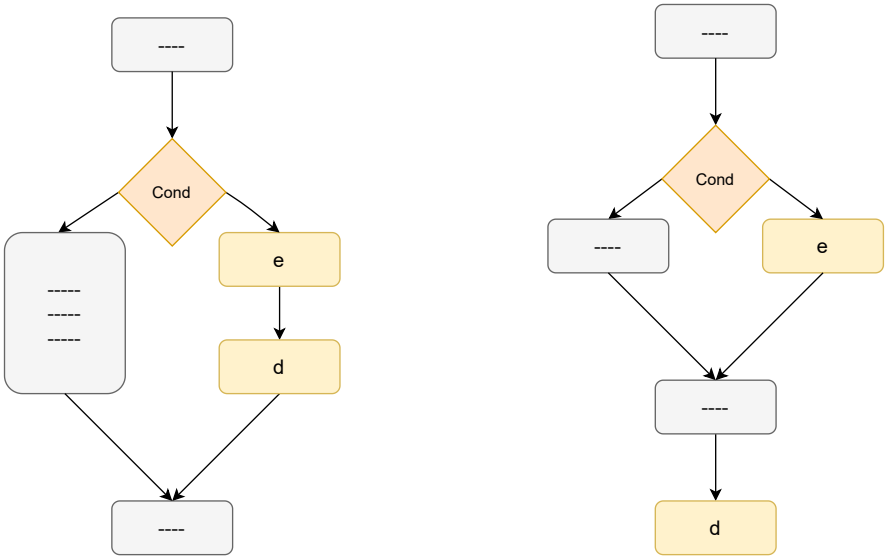


Figure 5.7: Cases where elimination of e is safe.

□

As far as read elimination goes, since we only need the information of read event that is to be eliminated, we do not need additional conditions to hold at the program level when conditionals exist. There is however, one case, in which the read itself is the conditional check. But the resultant code after elimination depends on the intention of the compiler, which can be either of 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 do not assume why the compiler would do such elimination, it is difficult to identify the target code and their resultant Candidate and Candidate Executions that are intended after such an elminiation. Hence we do not address this case and simply state that as long as the read to be eliminated is **not** a conditional check, it is safe to eliminate under the conditions of Theorem 5.1.

5.2.2 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 5.1, 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 write 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 state the following corollary.

Corollary 5.2.3. *Consider a program P having one loop with K representing the set of events within the loop. Consider appropriately candidates $C^1, C^2, ..., C^n$ each of which contain events e and d such that:*

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

Consider program P' after eliminating e. If

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

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

Proof. We need to show that in each iteration of some C^i the write e can be eliminated. For this, we need to have some write d that can exist in all candidates where e can exist. This condition corresponds to Corollary 5.2.2, which handles the case of conditonals too. The set of conditions for observable behaviors of the resultant program to be a subset is Corollary 5.2.1 being applied successively for each candidate C^i and all its iterations where e and d exist. By transitive property of subsets, we can infer that the resultant candidate C'^i has observable behaviors as a subset of original C^i. Doing this for all valid canddiates extends to program level by property of union of sets and subset relations[1].

□

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. Because a loop can have any number of iterations, we may have equal number of reads/writes as the iterations. The crux is that now we have proven when elimination is valid under the relaxed memory model. This in conjunction with reordering helps us determine the conditions under which loop invariant code motion can be valid. For each case, we use the same property that the resultant candidate execution, and hence candidate to program, the observable behaviors are a subset.

We first consider the case of reordering a Read outside a loop. There are two subcases for this, viz. one reordering the read before and one after the loop.

Corollary 5.2.4. *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*

$$e : uo \tag{5.6}$$

$$\forall k \neq e \in K,\ Reord(k, e) \tag{5.7}$$

$$\nexists C^i \in P \ s.t.\ e^{j \leq i} \notin C \tag{5.8}$$

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

[1]One might ideally think based on Theorem 5.2 that we could relax the constraint on the existence of d as there is another write e that would exist in the subsequent loop. This however, would rely on the fact that the last iteration of the loop in any execution of the program is guaranteed to have such an event d. This is not possible to ascertain as we do not have any information about the number of iterations of the loop. It may also be the case that the loop may never end.

Proof. We first consider the programs with just one iteration of the loop. Hence for Candidates of the form C^1, we have just e^1. We need to ensure that the resultant candidates C'^1 such that

$$\forall k \in K,\ e \overrightarrow{_{ao}}\ k$$

has observable behaviors as a subset of C^1

For every $k \in K$ such that $k \overrightarrow{_{ao}}\ e$ we need them to be reorderable with respect to e to bring it outside the loop. From condition 5.7 we have $Reord(k, e)$ for any such k. By Corollary 4.1.2, we can infer that C'^1 has observable behaviors as a subset of C^1. To extend this to the program level with one iteration, we also need that e should not be in any conditional branch. Condition 5.8, from Prop 1 implies that e is not part of any conditional branch. Thus, from Corollary 4.1.4, we can infer for program with one iteration of the loop that reordering outside the loop is safe.

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 possibly two reads e^1 and e^2. We need candidate C'^2 with just one such event e, such that:

  $$\forall k \in K,\ e \overrightarrow{_{ao}}\ k$$

  We first reorder both the reads e^1, e^2 to be outside the loop, naming it Candidate C''^2. Because we have Condition 5.7, by Corollary 4.1.2, 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 Condition 5.6, by Theorem 5.1, 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 5.7, we can infer

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

By Corollary 4.1.2, we can infer that C''^{n+1} with $cons(e^n, e^{n+1})$ has observable behaviors as a subset of C^{n+1}. From Condition 5.6, by Theorem 5.1, 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.

From Condition 5.8, by Corollary 4.1.4, we can infer that the observable behaviors of P' is a subset of P.

□

Corollary 5.2.5. *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*

$$e : uo \tag{5.9}$$
$$\forall k \neq e \in K,\ Reord(e, k) \tag{5.10}$$
$$\nexists C^i \in P \text{ s.t. } e^{j<=i} \notin C \tag{5.11}$$

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 $k \xrightarrow[ao]{} e$ we need them to be reorderable with respect to e to bring it outside the loop. From condition 5.10 we have $Reord(k, e)$ for any such k. By Corollary 4.1.2, we can infer that C'^1 has observable behaviors as a subset of C^1. To extend this to the program level with one iteration, we also need that e

should not be in any conditional branch. Condition 5.11, from Prop 1 implies that e is not part of any conditional branch. Thus, from Corollary 4.1.4, we can infer for program with one iteration of the loop that reordering outside the loop is safe.

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 possibly two reads e^1 and e^2. We need candidate C'^2 with just one such event e, such that:

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

  We first reorder both the reads e^1, e^2 to be outside the loop, naming it Candidate C''^2. Because we have Condition 5.10, by Corollary 4.1.2, 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 Condition 5.9, by Theorem 5.1, 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 5.7, we can infer

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

  By Corollary 4.1.2, we can infer that C''^{n+1} with $cons(e^n, e^{n+1})$ has observable behaviors as a subset of C^{n+1}. From Condition 5.9, by Theorem 5.1, 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.

From Condition 5.11, by Corollary 4.1.4, we can infer that the observable behaviors of P' is a subset of P.

□

Now we consider the case of reordering a write outside a loop, similar two subcases as for reads.

Corollary 5.2.6. *Consider K to be the set of events within a loop in program P. Consider e to be a write within the loop. Consider program P' with event e agent ordered before the loop. If*

$$e : uo \tag{5.12}$$

$$\forall k \neq e \in K,\ Reord(k, e) \tag{5.13}$$

$$\nexists C^i \in P \ s.t.\ e^{j<=i} \notin C \tag{5.14}$$

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 need them to be reorderable with respect to e to bring it outside the loop. From condition 5.13 we have $Reord(k, e)$ for any such k. By Corollary 4.1.2, we can infer that C'^1 has observable behaviors as a subset of C^1. To extend this to the program level with one iteration, we also need that e should not be in any conditional branch. Condition 5.14, from Prop 1 implies that e is not part of any conditional branch. Thus, from Corollary 4.1.4, we can infer for program with one iteration of the loop that reordering outside the loop is safe.

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 writes eads e^1 and e^2. We need candidate C'^2 with just one such event e, such that:

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

We first reorder both the reads e^1, e^2 to be outside the loop, naming it Candidate C''^2. Because we have Condition 5.10, by Corollary 4.1.2, 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 Condition 5.9, by Theorem 5.2, we can eliminate e^1, thus resulting in C'^2 whose observable behaviors is a subset of C''^{22}.

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 5.13 we can infer

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

  By Corollary 4.1.2, we can infer that C''^{n+1} with $cons(e^n, e^{n+1})$ has observable behaviors as a subset of C^{n+1}. From Condition 5.12, by Theorem 5.2, we can eliminate e^n, 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.

From Condition 5.14, by Corollary 4.1.4, we can infer that the observable behaviors of P' is a subset of P.

□

[2]Note that here, it is only safe to eliminate the first write (e^1) in contrast to having the choice to eliminate either e^1 or e^2 when both are reads.

Corollary 5.2.7. *Consider K to be the set of events within a loop in program P. Consider e to be a write within the loop. Consider program P' with event e agent ordered before the loop. If*

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

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 need them to be reorderable with respect to e to bring it outside the loop. From condition 5.2.7 we have $Reord(k, e)$ for any such k. By Corollary 4.1.2, we can infer that C'^1 has observable behaviors as a subset of C^1. To extend this to the program level with one iteration, we also need that e should not be in any conditional branch. Condition 5.2.7, from Prop 1 implies that e is not part of any conditional branch. Thus, from Corollary 4.1.4, we can infer for program with one iteration of the loop that reordering outside the loop is safe.

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. We need candidate C'^2 with just one such event e, such that:

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

  We also have from property of loops that $e^1 \xrightarrow[ao]{} e^2$. Having Condition 5.2.7, 5.2.7, by Corollary 5.2.1, we can eliminate e^1, giving us C''^2 whose observable

behaviors as a subset of C^2. From Corollary 4.1.3, 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 5.2.7, we can infer

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

  Having Condition 5.2.7, by Corollary 5.2.1, 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.

From Condition 5.2.7, by Corollary 4.1.4 and 5.2.2, we can infer that the observable behaviors of P' is a subset of P.

□

Reordering two events accross loops The above corollaries are defined for programs with one loop. However, they are valid for whenever loop invariant code motion is just involving removing an event outside one loop. In practice programs can have multiple loops, even those nested within each other and the compiler is free to reorder events accross nesting as well as from one loop to another. We cannot prove when we can reorder events accross two different loops or reordering events to be within a loop. At the Candidate level, this would imply reordering, elimination as well as **introduction** of events. Hence, this would require a proof of redundancy introduction right from the candidate execution level and then be extended to candidates and then to programs. This is beyond the scope of this thesis. We consider this as a part of future work.

To summarize, this chapter addressed the validity of elimination under the ECMAScript Memory Model. We first built a conservative proof for elimination of read, followed by write based on candidate executions. We later extended it to programs abstracted to the set of shared memory events. We then proved when loop invariant code motion is valid using both elimination and reordering at a candidate execution level. In the next chapter, we conclude this thesis, by discussing limitations, next steps, critique of the semantics of the model and ending with general foundational problems that need to be addressed in this domain of research.

Chapter 6

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 then give our critique of the model itself in terms of mixed size and tear-free factor of accesses. We finally conclude with possible future work in the domain of relaxed memory models.

6.1 Limitations/Advantages

Throughout this thesis, we dealt with program transformations using abstract set of events and partial order relations. Additionally, contrast to relying on operational semantics for our reasoning, we relied in a relatively simple and clear axiomatic semantics. This approach however, is not without its limitations. We elicit the key ones we think are important.

Separation of Concerns Our approach to program transformations avoids the complexity of the algorithmic complexity of program transformations. We also do

not assume why the compiler would perform such a transformation. While this benefits our analysis to be independant of any particular optimization, it may pose as a bottleneck while trying to incorporate our results in practice. As a simple example, it might turn out to be that sequentially, a conditional will always return *true*. This would mean that no candidate execution can have events from the $false$ branch. Hence the compiler might choose to reorder the events within the *true* branch outside. But as per Corollary (REORDERING CONDITIONALS), we do not allow any reordering outside the conditional, 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. On the other hand, having such a separation can help compiler writers see the impact of the model on the basic program transformations clearly, thus allowing them to write relatively correct program transformation algorithms.

Validity of Transformations is a conservative Discuss with Clark what does it mean for our approach to be sound but not complete. Or is it that conservative analysis is not about soundness of completeness at all?

It is important to note that our approach to validity of elimination and reordering is conservative. We do not assume anything about events that belong to other agents/threads. We only use information on the events involved in the program transformation and those that are *agent-ordered* between them. Hence, there could be several cases where one could reorder or eliminate events that we prohibit, but are still safe to do. From the perspective of the semantics of the memory model, this is possible because certain *happens-before* relations may not be relevant; they do not "trigger" any of the axioms of the model, if removed. Hence, such a transformation can be valid. This, however, as mentioned before, specific to certain programs. To keep track of such information while doing program transformation in practice would be infeasible as the program size increases.

Lack of Practical Results This work is purely theoretical. There is yet much more to be done to extend our results into practice. The main reason we resorted to first a theoretical guarantee is because literature has shown that mere empirical testing of results on concurrent programs is insufficient. Methods such as model checking, only work reasonably well for small programs. While this is another approach to identify counter examples to program transformations, it is infeasible in practice due to the sheer magnitude of possible candidate executions.

Mapping from Programming Constructs to Abstract events The specification and the results on program transformations are purely at the abstract set of shared memroy accesess. The concise mapping from ECMAScript's Read/Write(s) to these abstract events is something that must be done to extend our results in practice. As an example, we might have to perform aliasing analysis to identify which shared memory accesses are of same range. Such mappings however, are not required for our results; they do not influence it in any way.

6.2 Steps Further

We elicit in this section the complete roadmap we had in mind during hte inception of this thesis. We believe these are the following steps that anyone can take using our results to move towards practical relevance.

Addressing Read-Modify-Write So far we have assumed that no read modify write events exist in programs. However, this assumption is too strong in general (eg: Compare-and-Swap, Atomic Increament/Decrement are often used in programs). 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.

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

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

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

Addressing other basic program transformations Addressing redundancy introduction as an immediate next step would prove useful. Using it, we can analyze reordering of events accross loops. This will also give an interesting equivalence to instruction reordering. Other program transformations we found important to consider were strengthening/ weakening access modes (fence optimizations), gathering optimization, changing tearing factor of accesess.

6.3 Critique of the Model itself

Through the process of formalizing the memory model, we also critiqued the model in a few aspects. We here elicit mainly three of them which we consider to be due to underspecified semantics.

6.3.1 Tearing Factor and the Tear-free reads Axiom

The model states that all integer aligned accesses are tear-free. In terms of hardware, whether a memory access is tear free depends also on the bus-size. But if we still want to declare an access at the ECMAScript langauge level tear-free, the hardware must adopt some way to ensure that the access is indeed tear-free, meaning they respect the tear-free axiom. Alternatively, if at the ECMAscipt langauge level, we declare an access to tear despite the hardware having it to be tear-free, this wuold mean the compiler can perform certain aggressive optimizations to leverage this relaxation.

If we are using teared memory accesses, the set of observable behaviors becomes slightly non-trivial to justify. For instance, consider the program below, where we assume that x represents an 12-byte memory which is initialized to zero. Suppose that read to x is tearing but the writes are tear-free.

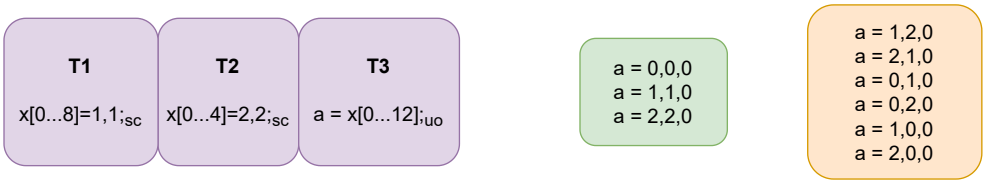


Figure 6.1: Mixed-size Tearing reads example

The green box represents the outcomes that we can quite intuitively justfiy. But the model also allows the outcomes in the orange box. This is because of the read that tears, which implies that Axiom 2 does not constrain $stck_{rf}$ to be functional

with respect to the read and the writes to the same 8-byte memory. Whether this is left to the hardware or program transformations is uncertain. If so, how could one justify any of the outcomes in the orange box is yet to be understood by us.

The main concern is that if both the writes are tear-free as well as sequentially consistent, how is it that the read can read them as if they are teared? Does this mean that the tearing factor of SC events depends on the existance of reads that tear or do not tear? If this is so, it would be non-trivial to reason about programs locally.

Consider the same last program above but with the read also of type *sc*. Yet the above two behaviors having those read values are allowed., Even Axiom 3 does not restrict such an observable behavior.

We believe this to be a problem with the tear-free reads axiom. A possible direction towards resolving this is to define the notion of tear-free / tearing using read-bytes-from relation.

6.3.2 Range of Initialize events uncertain

Whether the range of init events is the entire shared memory buffer or is it byte level granularity is uncertain. This affects the way we can reason with our programs and their corresponding observable behaviors.

Consider the two candidates below, each of which could represent the same program, one where the initialize event is to the whole 8-byte and one where its split into two events of type *init* writing 4 bytes each. The orange box below is an observable behavior in question.

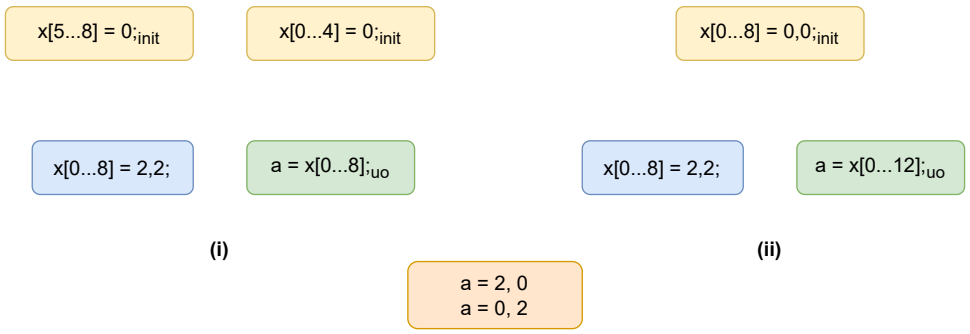


Figure 6.2: Mixed-size Init events example

For the first candidate, the outcome in question is possible. But for the second program, the outcome is not possible due to Axiom 2. Which one must correspond

to the original program is unclear and is not specificed by the semantics of the model specified in the standard.

6.3.3 Mixed-size events do not respect Coherence irrespective of access mode

Events that are unordered need not respect Coherence, unless constrained by happens-before relation. These accesses, mixed with sequentially consistent ones, give non-trivial behaviors.

For instance consider the program below.

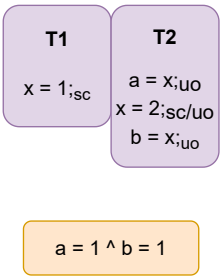


Figure 6.3: Coherence violation Example.

The above program can have the observable behavior under question. But this would imply that the value of write to x as 2, though local to the subsequent read, vanishes. The keen reader will note that such a behavior cannot even be explained by reordering or elimination of events. Note also that this observable is possible even if the write $x = 2$ has an access mode of *sc*.

Suppose we do have standard coherence respected. In this case too, sequentially consistent accesses which are mixed size give non-trivial behaviors. Consider another example with a few mixed size accesses, where the writes are of type *sc*.

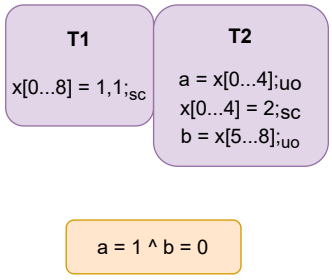


Figure 6.4: Mixed-size events example where coherence is assumed to be held.

Here the value of read $a = x[0...4]$, despite being 1 does not restrict the outcome of the read $b = x[5...8]$ to be 0. Again, we do not know how to justify this outcome using either reordering or elimination.

6.4 Future Directions in Weak Memory Consistency

We elicit certain problems that we consider to be foundational with respect to relaxed memory models. Having a concrete progress in any of these directions would in our eyes help solve problems concerning the specification of such models, compiler correctness and verification of programs with relaxed memory accesses.

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. 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 the same concern for high-level programming languages.

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 atomic accesses were semantically not defined. This may not be the case for hardware models such as ARMv8. Flur et.al [19] give their insights on having tested and observed mixed-size behaviors in hardwares.

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 ([19] is for an older version of ARM model).

Transformational Specification of Memory Models Using relaxed memory accesses certainly has a good impact on performance. Their semantics, however, are shown to be often not so suitable for quickly assessiing their impact on program transformations done by the compiler or the hardware. This thesis 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 our knowledge that the validity of program transformations still remains 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 using program transformations. Lahav et.al [36] try to do this for Total-Store-Order (TSO) and a fraction of Power memory model. They do this by describing transformationally over SC. 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. Given the increasing use of Heterogenous hardware, it would also be useful to have an automated process of constructing a memory model parametric to the choice of program transformations we would want to do.

Automation of Specification of Weak Memory Models This thesis also showcased 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. Lustig et.al **Lustig** do the reverse; they construct litmus tests parametric to memory consistency models. One could gain some insights from this work to do the reverse.

The advent of concurrent computation called upon a major change in how we view our programs execute, forcing us to move away from sequential reasoning. With the introduction of relaxed memory consistency, this sequential reasoning became more of a problem. The need for non-sequential reasoning is still not something that is appreciated by many, which is why we still have problems that come up with the specification of weak memory models. This thesis is a minor contribution towards having an intuitve non-sequential reasoning about concurrent programs, in a way that also in our eyes facilitates one to quickly identify potential problems with respect to program transformations. A major change at the educational as well as industrial level has to come, in order to propel the transformation towards user's view of their programs not as a sequential computation machines, but as a collection of partial orders with dependencies established between program components. We hope that this work helps in paving way towards that direction.

Bibliography

[1] Draft, "ECMAScript language specification," 2020. [Online]. Available: https://tc39.es/ecma262/#sec-memory-model.

[2] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, no. 9, p. 569, Sep. 1965, ISSN: 0001-0782. DOI: 10.1145/365559.365617. [Online]. Available: https://doi.org/10.1145/365559.365617.

[3] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Computers*, vol. 28, no. 9, pp. 690–691, 1979. DOI: 10.1109/TC.1979.1675439. [Online]. Available: https://doi.org/10.1109/TC.1979.1675439.

[4] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996. DOI: 10.1109/2.546611. [Online]. Available: https://doi.org/10.1109/2.546611.

[5] P. Sewell, "Memory, an elusive abstraction," in *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, J. Vitek and D. Lea, Eds., ACM, 2010, pp. 51–52. DOI: 10.1145/1806651.1806660. [Online]. Available: https://doi.org/10.1145/1806651.1806660.

[6] Nardelli, P. Sewell, J. Sevčík, S. Sarkar, S. Owens, L. Maranget, M. Batty, and J. Alglave, *Relaxed memory models must be rigorous*, 2009.

[7] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave, "The semantics of x86-cc multiprocessor machine code," in *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Z. Shao and B. C. Pierce, Eds., ACM, 2009, pp. 379–391. DOI: 10.1145/1480881.1480929. [Online]. Available: https://doi.org/10.1145/1480881.1480929.

[8] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: X86-tso," in *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., ser. Lecture Notes in Computer Science, vol. 5674, Springer, 2009, pp. 391–407. DOI: 10.1007/978-3-642-03359-9_27. [Online]. Available: https://doi.org/10.1007/978-3-642-03359-9%5C_27.

[9] Intel, "Intel 64 architecture memory ordering white paper," 2007. [Online]. Available: http://www.cs.cmu.edu/~410-f10/doc/Intel_Reordering_318147.pdf.

[10] W. Pugh, "Fixing the java memory model," in *Proceedings of the ACM 1999 Conference on Java Grande, JAVA '99, San Francisco, CA, USA, June 12-14, 1999*, G. C. Fox, K. E. Schauser, and M. Snir, Eds., ACM, 1999, pp. 89–98. DOI: 10.1145/304065.304106. [Online]. Available: https://doi.org/10.1145/304065.304106.

[11] J. Manson, "The design and verification of java's memory model," in *Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*, M. Ibrahim, Ed., ACM, 2002, pp. 10–11. DOI: 10.1145/985072.985078. [Online]. Available: https://doi.org/10.1145/985072.985078.

[12] J. Manson, W. Pugh, and S. V. Adve, "The java memory model," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, J. Palsberg and M. Abadi, Eds., ACM, 2005, pp. 378–391. DOI: 10.1145/1040305.1040336. [Online]. Available: https://doi.org/10.1145/1040305.1040336.

[13] J. Bender and J. Palsberg, "A formalization of java's concurrent access modes," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, 142:1–142:28, 2019. DOI: 10.1145/3360568. [Online]. Available: https://doi.org/10.1145/3360568.

[14] H. Boehm and S. V. Adve, "Foundations of the C++ concurrency memory model," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, R. Gupta and S. P. Amarasinghe, Eds., ACM, 2008, pp. 68–78. DOI: 10.1145/1375581.1375591. [Online]. Available: https://doi.org/10.1145/1375581.1375591.

[15] V. Vafeiadis, "Formal reasoning about the c11 weak memory model," in *Proceedings of the 2015 Conference on Certified Programs and Proofs*, ser. CPP '15, Mumbai, India: Association for Computing Machinery, 2015, pp. 1–2, ISBN: 9781450332965. DOI: 10.1145/2676724.2693181. [Online]. Available: https://doi.org/10.1145/2676724.2693181.

[16] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing C++ concurrency," in *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, T. Ball and M. Sagiv, Eds., ACM, 2011, pp. 55–66. DOI: 10.1145/1926385.1926394. [Online]. Available: https://doi.org/10.1145/1926385.1926394.

[17] K. Nienhuis, K. Memarian, and P. Sewell, "An operational semantics for C/C++11 concurrency," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, E. Visser and Y. Smaragdakis, Eds., ACM, 2016, pp. 111–128. DOI: 10.1145/2983990.2983997. [Online]. Available: https://doi.org/10.1145/2983990.2983997.

[18] O. Lahav, V. Vafeiadis, J. Kang, C. Hur, and D. Dreyer, "Repairing sequential consistency in C/C++11," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, A. Cohen and M. T. Vechev, Eds., ACM, 2017, pp. 618–632. DOI: 10.1145/3062341.3062352. [Online]. Available: https://doi.org/10.1145/3062341.3062352.

[19] S. Flur, S. Sarkar, C. Pulte, K. Nienhuis, L. Maranget, K. E. Gray, A. Sezgin, M. Batty, and P. Sewell, "Mixed-size concurrency: Arm, power, c/c++11, and SC," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, G. Castagna and A. D. Gordon, Eds., ACM, 2017, pp. 429–442. [Online]. Available: http://dl.acm.org/citation.cfm?id=3009839.

[20] C. Watt, C. Pulte, A. Podkopaev, G. Barbier, S. Dolan, S. Flur, J. Pichon-Pharabod, and S. Guo, "Repairing and mechanising the javascript relaxed memory model," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds., ACM, 2020, pp. 346–361. DOI: 10.1145/3385412.3385973. [Online]. Available: https://doi.org/10.1145/3385412.3385973.

[21] G. Naumovich and G. S. Avrunin, "A conservative data flow algorithm for detecting all pairs of statement that may happen in parallel," in *SIGSOFT '98, Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lake Buena Vista, Florida, USA, November 3-5, 1998*, ACM, 1998, pp. 24–34. DOI: 10.1145/288195.288213. [Online]. Available: https://doi.org/10.1145/288195.288213.

[22] S. P. Midkiff, J. Lee, and D. A. Padua, "A compiler for multiple memory models," *Concurr. Comput. Pract. Exp.*, vol. 16, no. 2-3, pp. 197–220, 2004. DOI: 10.1002/cpe.771. [Online]. Available: https://doi.org/10.1002/cpe.771.

[23] J. Sevcık and D. Aspinall, "On validity of program transformations in the java memory model," in *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, J. Vitek, Ed., ser. Lecture Notes in Computer Science, vol. 5142, Springer, 2008, pp. 27–51. DOI: 10.1007/978-3-540-70592-5_3. [Online]. Available: https://doi.org/10.1007/978-3-540-70592-5%5C_3.

[24] R. Morisset, P. Pawan, and F. Z. Nardelli, "Compiler testing via a theory of sound optimisations in the C11/C++11 memory model," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, H. Boehm and C. Flanagan, Eds., ACM, 2013, pp. 187–196. DOI: 10.1145/2491956.2491967. [Online]. Available: https://doi.org/10.1145/2491956.2491967.

[25] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Z. Nardelli, "Common compiler optimisations are invalid in the C11 memory model and what we can do about it," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, S. K. Rajamani and D. Walker, Eds., ACM, 2015, pp. 209–220. DOI: 10.1145/2676726.2676995. [Online]. Available: https://doi.org/10.1145/2676726.2676995.

[26] J. Sevcık, "Safe optimisations for shared-memory concurrent programs," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, M. W. Hall and D. A. Padua, Eds., ACM, 2011, pp. 306–316. DOI: 10.1145/1993498.1993534. [Online]. Available: https://doi.org/10.1145/1993498.1993534.

[27] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data mining for weak memory," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, Jul. 2014, ISSN: 0164-0925. DOI: 10.1145/2627752. [Online]. Available: https://doi.org/10.1145/2627752.

[28] P. S. Sindhu, J.-M. Frailong, and M. Cekleov, "Formal specification of memory models," in *Scalable Shared Memory Multiprocessors*, M. Dubois and S. Thakkar, Eds. Boston, MA: Springer US, 1992, pp. 25–41, ISBN: 978-1-4615-3604-8. DOI: 10.1007/978-1-4615-3604-8_2. [Online]. Available: https://doi.org/10.1007/978-1-4615-3604-8_2.

[29] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis, "Effective stateless model checking for c/c++ concurrency," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. DOI: 10.1145/3158105. [Online]. Available: https://doi.org/10.1145/3158105.

[30] M. Kokologiannakis, A. Raad, and V. Vafeiadis, "Model checking for weakly consistent libraries," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 96–110, ISBN: 9781450367127. DOI: 10.1145/3314221.3314609. [Online]. Available: https://doi.org/10.1145/3314221.3314609.

[31] C11, "Memory model," [Online]. Available: https://en.cppreference.com/w/c/language/memory_model.

[32] C. Verbrugge, A. Kielstra, and Y. Zhang, "There is nothing wrong with out-of-thin-air: Compiler optimization and memory models," in *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI '11, San Jose, CA, USA, June 5, 2011*, J. S. Vetter, M. Musuvathi, and X. Shen, Eds., ACM, 2011, pp. 1–6. DOI: 10.1145/1988915.1988917. [Online]. Available: https://doi.org/10.1145/1988915.1988917.

[33] Arvind and J. Maessen, "Memory model = instruction reordering + store atomicity," in *33rd International Symposium on Computer Architecture (ISCA 2006), June 17-21, 2006, Boston, MA, USA*, IEEE Computer Society, 2006, pp. 29–40. DOI: 10.1109/ISCA.2006.26. [Online]. Available: https://doi.org/10.1109/ISCA.2006.26.

[34] D. Marino, A. Singh, T. D. Millstein, M. Musuvathi, and S. Narayanasamy, "DRFX: a simple and efficient memory model for concurrent programming languages," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, B. G. Zorn and A. Aiken, Eds., ACM, 2010, pp. 351–362. DOI: 10.1145/1806596.1806636. [Online]. Available: https://doi.org/10.1145/1806596.1806636.

[35] J. Kang, C. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer, "A promising semantics for relaxed-memory concurrency," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, G. Castagna and A. D. Gordon, Eds., ACM, 2017, pp. 175–189. [Online]. Available: http://dl.acm.org/citation.cfm?id=3009850.

[36] O. Lahav and V. Vafeiadis, "Explaining relaxed memory models with program transformations," in *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, J. S. Fitzgerald, C. L. Heitmeyer, S. Gnesi, and A. Philippou, Eds., ser. Lecture Notes in Computer Science, vol. 9995, 2016, pp. 479–495. DOI: 10.1007/978-3-319-48989-6_29. [Online]. Available: https://doi.org/10.1007/978-3-319-48989-6%5C_29.