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

Concurrent memory accesses have been shown to give us tremendous performance benefits compared to their sequential counterparts. With the recent addition of several hardware features such as read/write buffers, speculation, etc., more efficient forms of concurrent memory accesses are introduced. Called relaxed memory accesses, they are used to gain substantial improvement in the performance of concurrent programs. A relaxed memory consistency model specifically describes the semantics of such memory accesses for a particular programming language. Historically, such semantics are often ill-defined or misunderstood, and have been shown to conflict with common program transformations essential for the performance of programs overall. In this thesis, we give a formal declarative(axiomatic) style description of the ECMAScript relaxed memory consistency model. We analyze the impact of this model on two of the most common program transformations, viz. instruction reordering and elimination. We give a conservative proof under which such optimization is allowed for relaxed memory accesses. We use this result to reason about the validity of loop invariant code motion under the same model. We conclude this thesis by eliciting the limitations of our approach, critique on the semantics of the model, possible future work using our results, and pending foundational questions that we discovered while working on this thesis.

Abrégé

Acknowledgements

There are countless people who have influenced me through these years and have motivated me to keep doing this thesis. Unfortunately, I cannot mention all of their names due to space constraints as well as the fact that I may not know their influence on me yet. However, I do want to mention a few important names and people that have in my eyes influenced me the most. Firstly, my advisor Professor Clark Verbrugge. Having not done any rigorous theoretical work or research before and yet wanting to do a theoretical one as a thesis, he gave me high level guidelines to ground me and to keep me being overwhelmed from framing my own theorems, lemmas, proofs, etc. I would not have gotten a better advisor than him, especially during the initial years when I was trying to build an intuitive foundation. Also, to advise someone in this sense is quite difficult in my eyes. Secondly, my friend, Aarti Kashyap, mainly for being a patient listener, a motivator that made me attend several conferences and for always giving constructive feedback. I needed an outside perspective and somebody to discuss with about various aspects of this research at any time; and she being there for it has helped me shape ideas of this thesis better. Thirdly, Conrad Watt, mainly for giving me an inside perspective of this topic of research and motivating me to pursue this thesis topic. My family, friends and lab-mates, who always assured me the support (mentally or otherwise) I would need at any time during my masters. I would like to also thank Viktor Vefeiadis, my internship advisor, who helped me look at the research I did all this while in a simplistic way. This has in some way reflected the way I write or present my research ideas. Finally, God, for guiding me in his/her own way and presenting me with ample opportunities to grow as a researcher as well as a human being these years.

Contents

List of Figures

Chapter 1

Introduction

With the increasing demand for performance in computing, the use of concurrency has become quite ubiquitous. Particularly, the use of *shared memory* concurrency 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 performance. The use of such accesses, unmonitored can cause programs to mis-behave, or can cause quite a bit of slowdown. The lack of well-defined semantics which is generally observed for them make it difficult for the user to utilize these accesses more responsibly. In addition, usage of such accesses may invalidate several aggressive optimizations that the compiler does which contributes to a major chunk of our desired performance.

In current times, there is an increasing trend/need to offload computations on the web. One major benefit of this is that portability issues are reduced greatly, which means I can run complex softwares 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 comparable with those done offline, performance plays a major factor, and this factor has always pushed people to not rely on web-based computations.

In this scenario, the use of shared memory access (more specifically, relaxed accesses) naturally fits in to provide the lack of performance we so desperately need. Recently, ECMAScript, the standard for all such web-scripting languages defined their own memory consistency model. Its semantics, however, is written in prose format. besides, no formal proof about the validity of common program transformations under such a semantics is even thought about. There are just suggestions for programmers in this respect, which are example driven. To add, this model also described semantics for mixed-size accesses, something which is quite recent and not precisely defined 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 behaviors (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 behaviors. We use this insight to prove validity of a subset of *loop-invariant code-motion*.

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* among events that constitute a program. We use these orders to define a program *execution* and its *outcomes*(referred 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* using program executions. 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 under-specification 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 should be addressed/investigated in the immediate future.

Specific contributions of our work include the following:

- 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.
- We show when basic reordering of independent instructions is allowed. We extend this to reordering in the presence of conditionals and loops in programs.
- Similar proof designs are used to validate other basic optimization behaviors such as removing redundant reads or writes. Further extending it to elimination in the presence of conditionals and loops.
- We 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* on memory consistency models coupled with relevant related work on its semantics, program transformations and other important problems.
- Chapter 3 give the formal *declarative style* semantics of the ECMAScript memory model.
- Chapter 4 dives into *instruction reordering*, with a formulation of relevant *lemmas, 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 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 important pending problems in this domain(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 [2]. 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 concerning validity of program transformations. Finally, we end with a list of tutorial works done in the axiomatic style specification of memory models.

A lot of problems emerged with the introduction of concurrent computing. Starting with the famously known mutual exclusion problem way back in the 1960s introduced and solved by Djikstra et al. [3], 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 revolves around the so-called "critical section"; which requires that every thread trying to do something in a shared memory region must have exclusive access to it before doing anything. In recent decades, the need for more performance has increased; multiple hardwares having different features such as read-write buffers, caches, etc exist. Reliance solely on critical sections to perform operations on shared memory is today observed as a bottleneck. Without its use, however, the possible outcomes of a concurrent program increases and complicates the notion of *atomicity* in actions done on shared memory.

2.1 Memory Consistency Models

In its essence, Concurrent programs take advantage of *out-of-order* execution. Intuitively, this means that more than one unrelated computation 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 of which are described by a *memory consistency model*.

Sequential Consistency(SC), which was first formulated by Lamport et al. [4], gives programmers a very intuitive way to reason about their programs running in a multiprocessor environment, with or without the use of critical sections. SC guarantees that every outcome of a program must be equivalent to a *sequential interleaving* of each thread's individual actions. For example, consider the program in Figure 2.1 with two threads, which share memory denoted by x, y initialized to 0, where a, b are local variables. The green box represents the possible values that a and b can read under sequential consistency rules.

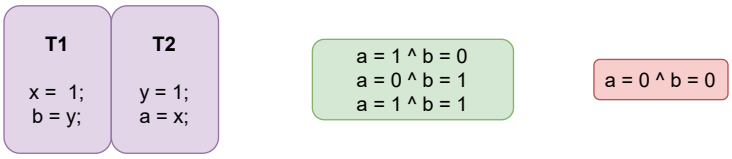

Figure 2.1: Example program with its allowed and disallowed outcomes under SC.

However, the above program under SC cannot have the outcome $a = 0 \ \wedge \ b = 0$ as given in the red box. 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. From a program transformation perspective, *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, redundancy elimination, introduction(a consequence due to optimizations such as *register allocation, partial expression elimination, etc.*) done by the hardware or the compiler. A tutorial by Adve et al. [5], 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 several executions of our programs. Hardware

features relax the sequential reasoning in programs, and hence models describing their semantics come to be known as relaxed memory models. Surprisingly, quite a few relaxed memory model specifications of different hardware/high level programming languages are written in informal prose format, which lead to a number of problems in implementation as mentioned in Sewell et al. [6]. A position paper by Nardelli et al. [7] summarizes these problems well. A lot of academic research revolves around addressing such problems, the bulk of which are centered around the memory models of Intel's x86, C11 and Java.

x86 Intel's white paper on their earlier versions of the memory model for x86 was described in their white paper[10]. Sarkar et al. [8] showed that the then x86 model was fairly informal, which they formalize in their x86-CC memory model. Owens et al. [9] later showed that the x86-CC model did not reflect precisely what x86 hardware intended, followed by exposing more problems in the latest specification of the x86 model given in the white paper of Intel [10]. They then propose a new model, x86-TSO as a remedy to this. The series of work on this exposed repeated inconsistencies between the specification and the implementation in hardware.

While memory consistency models were initially only concerning hardware languages, as the years went by, as the user level threading libraries started to come up, the need to have threads and its concurrent semantics as part of the high level language became apparent. From the relaxed memory consistency perspective, this was not an easy process; several changes and edits are still made to memory models of Java and C11 to date, bulk of which addressed concerns about informal/ill-defined specifications that made the model too vague for programmers to rely on.

Java Pugh et al. [11] showed the complexity of the initial versions of Java memory model, showing with seemingly simplistic examples that even those involved in its inception were not sure about its specification. Later, Manson et al. [12] also exposed many limitations and under-specified semantics within the model, for which they proposed a new model with concise semantics. Manson et al. [13] 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*(these concepts will later be discussed in the last section of this chapter). Recent works such as that done by Bender et al. [14], 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. [15] exposed that the earlier version of C++ concurrency semantics was unsound, followed by proposing a new semantics for the same.

Vafeiadis et al. [16] summarized the open problems and verification results in the C11 memory model. Some of these problems have been addressed, while some still remain open to date. Batty et al. [17] exposed the lack of clear specifications of the C11 memory model, giving a clarified, mathematical specification of the model. This work really helped change the standard specification format of C11 memory model which is found today[34]. Nienhuis et al. [18] gave an operational semantics of the then C11 memory model, exposing certain limitations in terms of execution of such concurrent programs. They discovered that one cannot build an equivalent operational model to respect sequentially consistent and synchronization order (defined by the C11 memory model). Lahav et al. [19] demonstrated that the current compilation scheme of C11 concurrent programs to POWER unsound, and proposed fixes to the model, 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 128-bit memory accesses that can be done either atomically or split across different subsequent memory accesses. Investigation of semantics in this direction is fairly recent. Flur et al. [20] investigated mixed-size behaviors in Arm and POWER architectures, also exposing new problems to address in the semantics such as ordering between mixed size events in an execution. 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 [21] uncovered and fixed a deficiency in the previous version of the model, repairing the model to guarantee SC-DRF(again, defined in the last section of this chapter). Our analysis is based on this corrected model 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.

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, 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. A large class of optimizations are unsound under concurrent programs employing a memory model such as SC. With the introduction of constraints on relaxed memory accesses, quite a few critical program transformations responsible for huge performance gains became possible, but were hard to prove it valid in general.

This situation necessitates for a fundamental change in how we employ data-flow analysis to optimize programs in a concurrent context. Naumovich et al. [22] proposed one such analysis which is today commonly known as May-Happen-In-Parallel analysis. Midkiff et al. [23] 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, required 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.

For data-flow analysis, a recent work by Alglave et al. [24] show that non-relational data-flow analysis are sound under a category of relaxed memory models. But code transformations are themselves not yet investigated with each one in its own respect, which entail a number of data-flow information and code motions.

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. We start by showing how using counterexamples one can show basic transformations are invalidated under SC.

For instance, the disallowed outcome in Figure 2.1 should be possible; we can simply reorder either the two events in $T1$ or those 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 is an example where we would think it is okay to reorder two independent reads.

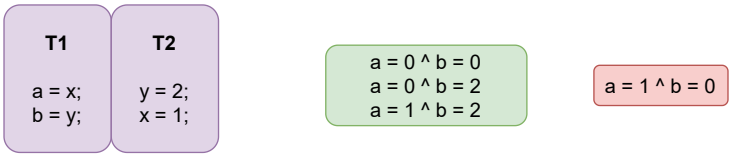

Figure 2.2: Program where reordering of independent reads seems possible with its allowed(green box) and disallowed(red box) outcomes in SC.

The reordered program can justify a sequential interleaving of events to have the disallowed outcome in the original program. This is shown in Figure 2.3.

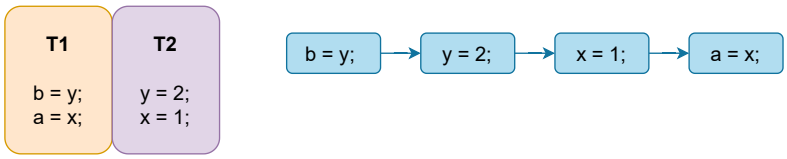

Figure 2.3: The transformed program reordering two reads in T1, with a trace justifying the disallowed outcome under SC.

Such concerns are not only related to reordering, but also elimination. Consider the program in Figure 2.4.

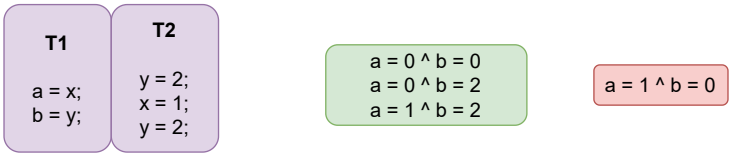

Figure 2.4: Program with a redundant write to y with its allowed(green box) and disallowed(red box) outcomes under SC.

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 2.5, can justify the outcome under SC which is disallowed in the original program.

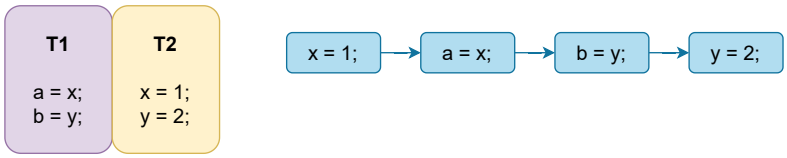

Figure 2.5: The transformed program eliminating the first $y = 2$, with a trace justifying the disallowed outcome under SC.

The above examples show that even simple transformations can be unsound under SC. Complex program transformations such as register allocation, common-sub-expression-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 variety of optimizations without breaking the consistency rules under which the concurrent program is supposed to behave/execute. In addition, hardware at that time had come up with several features as mentioned above, which in principle could be used effectively for performance, but were not so useful for programs respecting 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. [25] showed that standard compiler optimizations were rendered invalid under the respective memory model of Java. Simple transformations such read-after-write 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. [26] showed the soundness of optimizations with non-atomic memory accesses in the C11 memory model. Vafeiadis et al. [27] 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 committee for C11.

With respect to instruction reordering and redundancy elimination in shared memory programs, Ševčík et al. [28] 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 behavior. Our approach is in fact the other way round; we show that the optimized program does not introduce new behaviors, 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 races 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 specification. 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. [29]. 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. [30] specifies an axiomatic framework for specifying the behaviors of shared memory multiprocessors. Owens et al. [9], Batty et al.[17] specify the respective memory models of x86 and C11 in an axiomatic format.

2.4 Other Concerns

Up to 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 concern of compilation; whether the compilation(codegen phase mainly) from source and target with different memory models is correct? This question is quite open-ended given a varying class of memory models that exist. While we did not do literature review in this direction particularly, works done by Lahav et al. [19] and Watt et al. [21] on the C11 and JavaScript memory model respectively exposed incorrect compilation done due to which the program exhibited unintended behaviors on execution in particular hardwares (POWER and ARM respectively). The interested reader can refer to Podkopaev et al. [31] to explore further in this direction.

Verification/Model-Checking While the focus of this thesis is not on Formal Verification, there certainly does exist several research works in verifying weak mem-

ory programs. The main concern is that because of weak memory accesses, a program has even more several outcomes (which increase more exponentially compared to concurrent programs performing operations on shared memory only in *critical sections*.) One of the well known methods in this decade 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. [32] provided a model checker named RCMC for performing effective model checking of relaxed memory programs in C11. Michalis et al. [33] provided a model checker named GenMC for performing effective model checking parametric to relaxed memory models.

Out-of-Thin-Air A few well known 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 [34] 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 one may conclude that *out-of-thin-air* is a bad property that must be avoided at all costs, Verbrugge et al. [35] shows that disallowing them also disallows quite a few compiler optimizations.

Custom Memory Models We also explored certain research works that came up with their own memory model in order to simplify/solve the problems above. Arvind et al. [36] 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. [37] 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. [38] 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.

As a summary, this chapter elicited the key research works done in the domain

of relaxed memory models, its specification 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 specification of the same.

Chapter 3

The Memory Model

This chapter starts with exposing some problems with the existing specifications of the model. We then give our declarative specification of the model. 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 specify the *axioms* of the model. Lastly, we define *races* followed by defining what a *Consistent Execution* is as per the specification of the model.

The model we consider is the current draft specification [1] of the ECMAScript standard. The semantics of the model we consider has remain unchanged since the time we started our investigation (2019), so we believe our work will also be of use to those working on it. The specification is claimed to be *axiomatic* by definition, which should, in our view remove the complexities of the rest of the standard from the semantics of the model. However, there are some concerns with it:

The Model is Quite Algorithmic Although the standard states that the model is not supposed to be operational, the specifications of the model hint otherwise. They are defined as relational constraints on certain *abstract operations* which are not necessary to understand the semantics of the model. As an example, consider one of the *axioms* of the model in Figure 3.1 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 ECMAScript specification for Coherent Reads.

The definition in Figure 3.1 is specified in terms of a return value from an abstract operation. Understanding this requires one to know the definitions for *Ws, execution, SharedDataBlockEventSet*, etc. although this is not required to understand what 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 constraints using binary relations that we

derive(also in some sense, take directly) from the specification in Section 2 of this chapter.

Certain Unnecessary Definitions Certain abstract operations are not required to capture the semantics of the model. One such example is shown in Figure 3.2

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 ECMAScript specification for Compose Write Event Bytes [1].

Figure 3.2 is the definition of an abstract operation. Understanding this operation would require the meaning of the terms *ModifyOp, Payload, Ws* and *ByteIndex*. In its essence, this operation determines the read-values read by a single event by collecting the values from their corresponding writes. We noticed that one need not

know this operation nor understand its function as it does not play a role in the semantics of the model. Other such abstract operations which may not be essential are *ValueOfReadEvent* and *ValidChosenReads*[1].

Still a bit verbose The entire model, though algorithmic in its structure, is still quite verbose in its details, which makes it difficult to understand the model semantics. Figure 3.3 is another *axiom* from the standard.

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 ECMAScript specification for Sequentially Consistent Atomics axiom.

The definition in Figure 3.3, is not concise enough to reason about it mathemat-

ically. In addition, the part after Note1 in Figure 3.3 is not a semantic specification, rather a programming guideline while using atomic memory accesses. We reduce the above entire axiom into three main patterns using binary relations.

Given the above concerns about the specification in the standard, we found the need to have a concise formal description of the model. In the following sections, we define what agents and events are, followed by several binary relations among different events.

3.1 Agents

Agents represent threads in a concurrent program. As per the standard, they have more meaning[1] 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 may or may not belong to 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 modeled 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 Sets

Given an agent cluster, an *event set* $\boldsymbol{E}$ is a collection of all events from the agent event lists. This set is mainly composed of 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* events.
- **Synchronize (*S*) Events** These events only restrict the ordering of execution of events by agents. For instance *lock* and *unlock*[2]type of events would be categorized under Synchronize events.

Figure 3.4 summarizes the different subsets of ***E***.

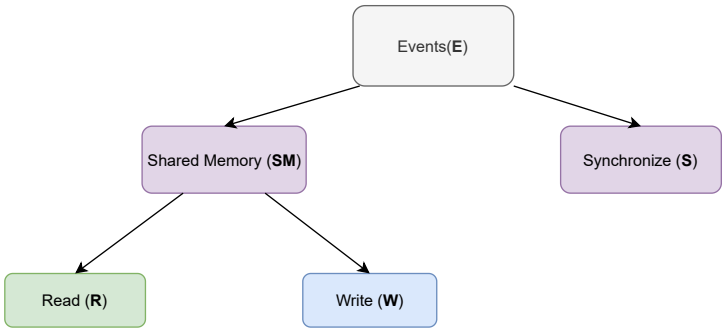

Figure 3.4: Different sets of events.

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 a function that maps a shared memory event to the range it operates on[3]. This we represent as a starting index i and a size s. For instance, we could represent the range of a write event w as

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

We define the two binary operators below on ranges as follows:

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

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

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*, partially *overlapping* or *equal*. We use the binary operators to define these three possibilities between ranges of events e and d :

1. Disjoint $\Re(e) \cap_{\Re} \Re(d) = \phi$

2. 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. Overlapping $(\Re(e) \cap_{\Re} \Re(d) \neq \phi) \wedge \neg(Equal(e, d))$

3.2.3 Event Order/Event Types

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[34]) 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* in nature[4]. 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 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 summarizes the above hierarchy diagrammatically.

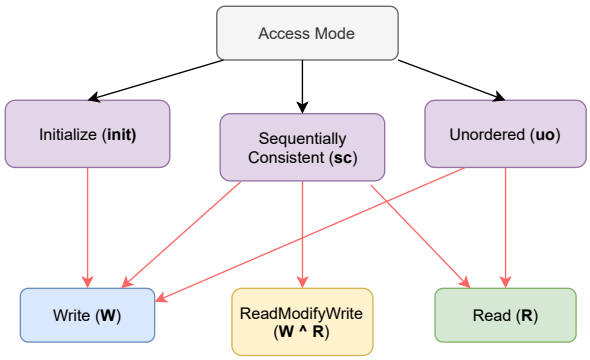

Figure 3.5: Event access modes with its restriction for some events.

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. Events 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* or *event* : *tf*. When representing events in examples, the type would be represented as a subscript: $event_{tf}$ or $event_{!tf}$.

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 e with range $(i, 3)$ and corresponding write events $d1$ and $d2$ with ranges $(i, 3)$ and $(i, 4)$ respectively. Possible $\overrightarrow{rbf}$ relations could be

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

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

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

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 examples, 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 3.6 below is an example of a program with its outcome (read values) shown in terms of reads-from relations. The left is a program example with the orange box representing one outcome of a program.

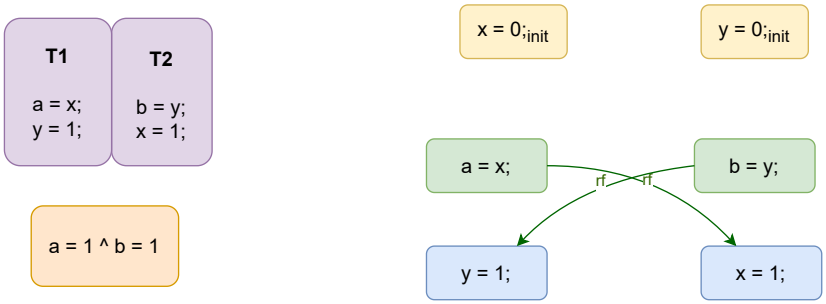

Figure 3.6: An example showing *reads-from* relations.

3.3.2 Agent-Synchronizes-With (ASW)

This is a set 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:

$$ASW_k = \{\langle s1, s2\rangle, \langle s3, s4\rangle ...\}$$

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].

$$\langle s1, s2\rangle \in ASW_k \Rightarrow s2 \notin ael(k).$$

The ordered tuples are acyclic.

$$\langle s1, s2\rangle \Rightarrow \neg\langle s2, s1\rangle.$$

Figure 3.7 below shows an example of this relation among two agents. The left represents a program with synchronize events $s1$ and $s2$. The orange box represents the synchronization established between the two agents.

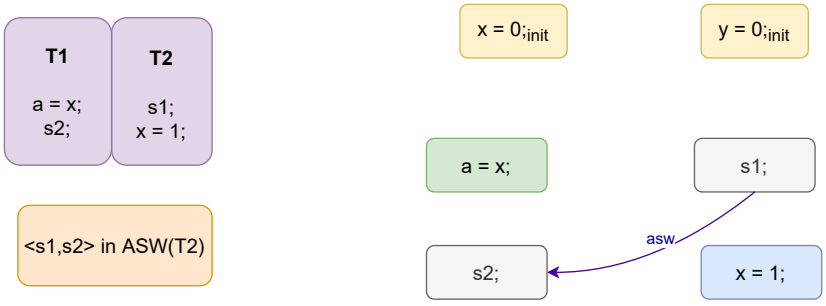

Figure 3.7: An example showing *asw* relations.

3.4 Ordering Relations among Events

We define agent-order, synchronize-with order, happens-before order and memory order using the binary relations defined on events in the previous section.

[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.

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

This is a union of the total orders among events belonging to the same agent event list. It is analogous to *intra-thread* ordering. For example, if two 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 shows an example of agent order between events(right) composing a program(left).

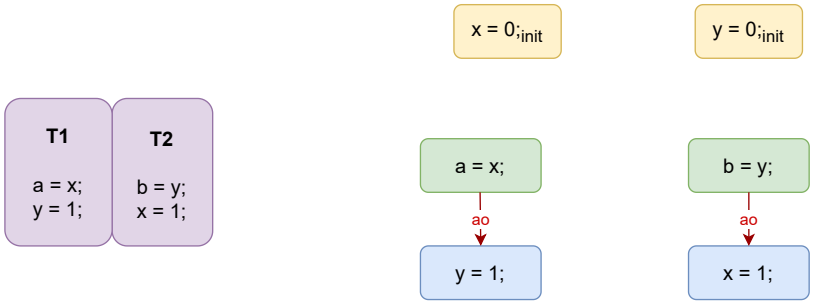

Figure 3.8: An example with *agent-order* among events.

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

This is a 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.

$$\langle e_i, e_j \rangle \in ASW \Rightarrow e_i \overrightarrow{sw} e_j.$$

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 shows examples of such orders that can exist between events(right) of a program(left). The orange box represents the agent-synchronizes-with set that exists coupled with a possible outcome of the program (the final read value of a).

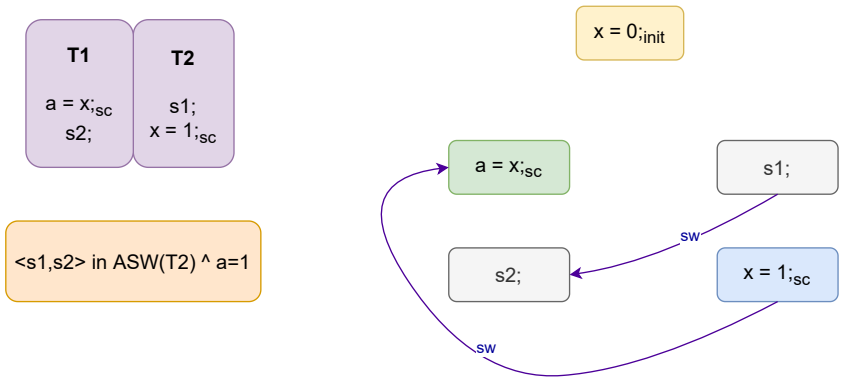

Figure 3.9: An example with *synchronize-with* relations among events.

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

This is a transitive order on events, composed of the following:

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

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

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

$$(e \overrightarrow{sw} d) \Rightarrow (e \overrightarrow{hb} d).$$

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

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

Figure 3.10 summarizes all possible patterns of happens-before order between events(right) of a program(left). THe orange box, again represents the agent-synchronizes-with set that exists coupled with a possible outcome of the program (the final read value of a).

[7]Note that for the second condition, both ranges of events have to be equal. This however, does not mean that the read cannot read from multiple write events(refer the definition for $\overrightarrow{rbf}$ in Subsection 3.3).

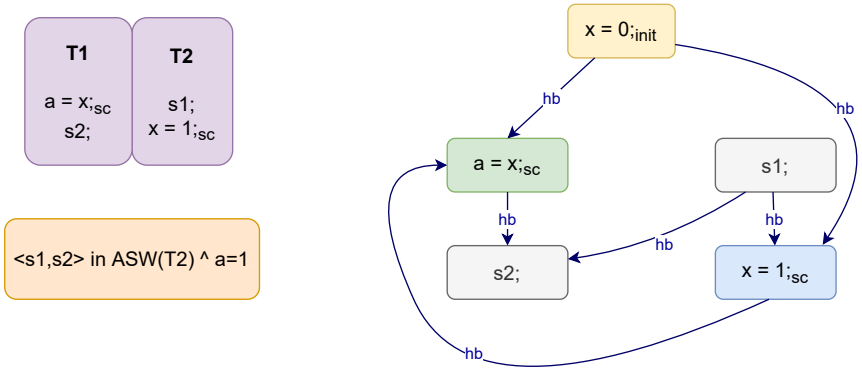

Figure 3.10: An example with all the variants of *happens-before* relations between events.

3.4.4 Memory Order ($\overrightarrow{mo}$)

This is a *total order* on all events that respect happens-before order[8].

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

Figure 3.11 is an example of a memory order between events(right) of a program(left).

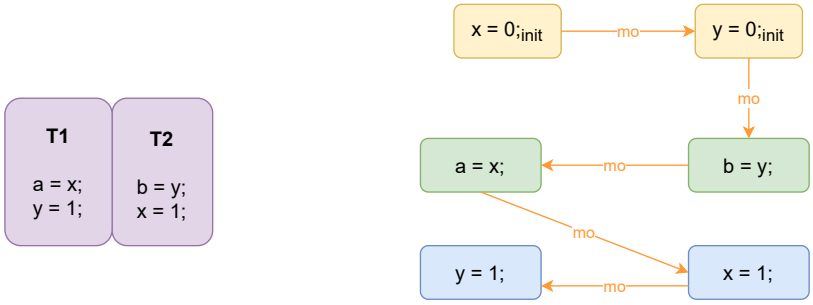

Figure 3.11: An example with *memory-order* (total) among all events.

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

[8]An interesting part is that memory order, though total, is a bit undefined as to how it weaves together this total order given different events.

ordering relations defined above. This will help us understand where the consistency rules actually apply.

Definition 3.5.1. *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. *A* candidate *is a collection of abstracted set of shared memory events of a program involved in one possible execution, ordered by 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. Figure 3.12 is an example of a Candidate.*

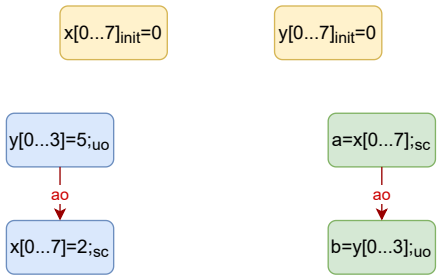

Figure 3.12: An example of a Candidate.

Definition 3.5.3. *A* candidate execution *is 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. Figure 3.13 shows an example of a Candidate Execution of a program with two threads, one writing to x and y an the other reading from it.*

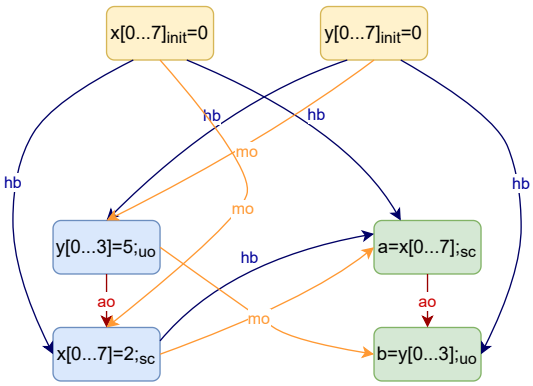

Figure 3.13: An example of a candidate execution based on candidate in Figure 3.12.

Definition 3.5.4. *A* observable behavior *is 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 shows an example of an observable behavior based on the candidate execution in Figure 3.13.*

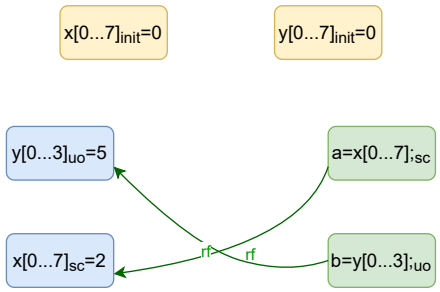

Figure 3.14: An example of observable behavior.

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 either overlapping or equal 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*

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

Figure 3.15 pictorially depicts the pattern based on the rule where e *cannot read from* d*. The represents the two binary relations between the reade and the writed that should not exist together.*

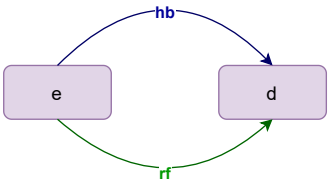

Figure 3.15: First axiom of Coherent Reads.

- *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 is on the rbf relation among two events.*

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

  Figure 3.16 pictorially depicts the pattern where e cannot read certain bytes from d. Such a pattern cannot exist given that the rbf relation from e to d is to a byte address common to d, g and e.

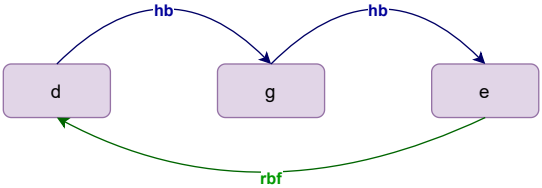

Figure 3.16: Second axiom of Coherent Reads where the pattern above cannot exist for the common range between d, g and e.

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]*.*

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

Figure **??** *shows the pattern that is disallowed among all tear-free events.*

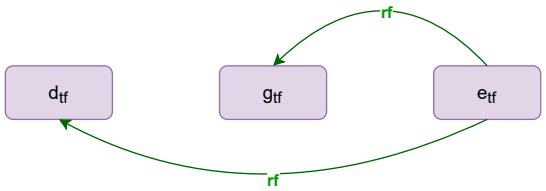

Figure 3.17: Axiom of Tear-free reads given that events e, g and d have equal ranges.

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 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.*

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

  Note that here, hb relation plays no role. Figure 3.18 depicts pictorially the pattern that is not allowed by this rule.

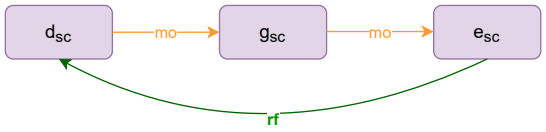

Figure 3.18: First Axiom of SC-Atomics.

- *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.*

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

[9] 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 32-bit system). We need a shared memory access to appear "tear-free", whenever it is declared to be so, irrespective of what hardware the program is run on.

Note that in this rule, it is not necessary to have a hb relation between d and g. Figure 3.19 pictorially pictorially the pattern that is not allowed by this rule.

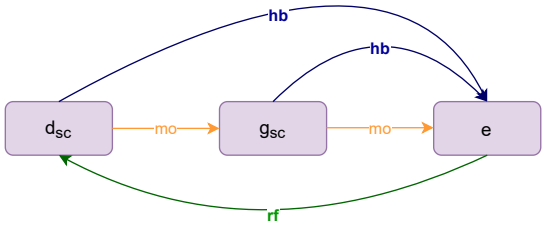

Figure 3.19: Second Axiom of SC-Atomics.

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

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

Note that here, it is not necessary to have a hb relation between g and e. Figure 3.20 pictorially depicts the pattern that is not allowed by this rule.

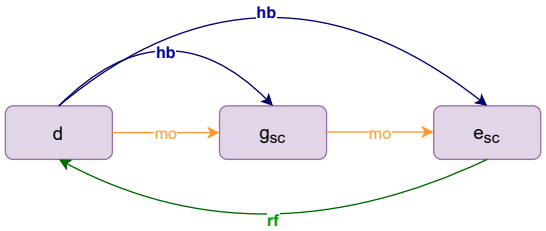

Figure 3.20: First Axiom of SC-Atomics.

3.7 Race

We would want our concurrent programs to be written in such a way that the order in which shared memory is read from or written to is deterministic. This, however, may not always be the case. For applications such as client-server interactions, we would rather be more comfortable keeping the client requests being serviced in a non-deterministic fashion. This is because clients can be in millions, and always having a fixed order in which they should be serviced may not be ideal. The accesses to shared memory that are involved, which need to be deterministically ordered, but

are left to program execution, are said to *race* and are in a *race condition*. Pairs of accesses that need to be ordered are those whose order affects the final outcome (observable behavior) of the program. In our view, they are simply those pairs, of which at least one of them is a write.

In the context of relaxed memory accesses, another type of race exists. In the normal *race condition*, though accesses are not ordered deterministically by us, every execution will give us an outcome that can be justified using one particular order in which the concurrent accesses were issued. However, there could be cases, i.e. outcomes of program execution, where no ordering among these accesses can justify[10]. In such a situation, concurrent accesses are said to have a *data race* among them.

We define the two forms of races formally with respect to the ECMAScript memory model.

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 when they are shared memory events:

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

having overlapping or equal ranges, not ordered by the $\overrightarrow{hb}$ 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:

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

[10]This is attributed to the notion of *atomicity*. Atomic accesses, ensure that only the outcomes that result due to their different orderings are possible. Other accesses that do not respect this rule are called *non-atomic*. In the context of our model, they are considered to be of type *uo*.

Data-Race-Free (DRF) Programs An execution is considered data-race-free if none of the above conditions for *data-race* occur among events in a candidate execution and its observable behaviors. 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[11]:

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 behavior.

As a summary, this chapter axiomatically defined the ECMAScript memory model. We defined binary relations on events and specified the constraints of the model in terms of restricting reads-from relations 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.

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 a few more definitions for our purpose, followed by two 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 reordering at a Candidate Execution level. Lastly, we address reordering at the program level involving conditional branching and loops. We use counter examples to give a better intuitive understanding of the elements of the proof.

4.1 Introduction

Instruction reordering is a common transformation done by the compiler/hardware, which is essential to optimizations such as instruction scheduling, loop invariant removal, partial redundancy elimination, etc. 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. The relaxed memory model of ECMAScript prescribes 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. In all the examples, we only show those ordering relations that are relevant to our purpose.

Consider the first example in Figure 4.1 below of a Candidate before and after reordering two events. The original candidate is to the left and that on the right is after reordering the two reads of $T2$. The observable behavior in question is written in the middle(orange box).

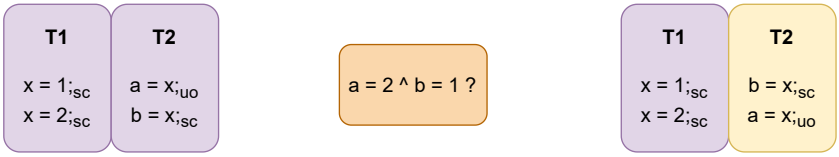

Figure 4.1: First example for reordering in candidates of the original program and its reordered counterpart.

Figure 4.2 has two sets of relations. The first justifies the observable behavior for the reordered Candidate. While the second provides justification for the original Candidate.

Notice that in the second set of relations, one may have a Candidate Execution with the mentioned memory order, where the read $(a = x)$ can read from a write $(x = 2)$ memory ordered after it. Neither Axiom 1 nor Axiom 3 restrict the pattern prescribed by this. This is quite counter-intuitive to understand at first. From the semantics of the model, this justification to the observable behavior however, is completely valid[1].

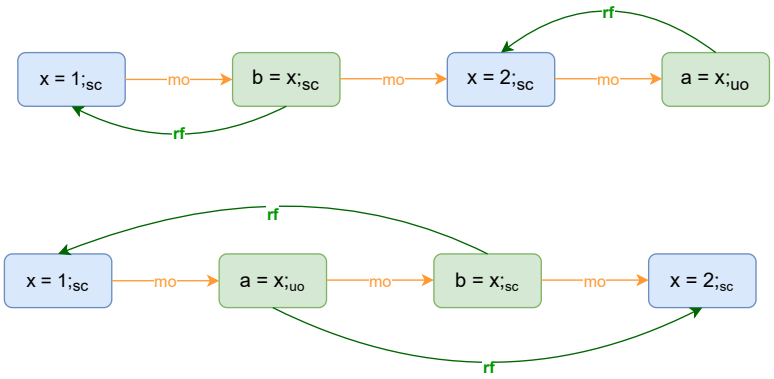

Figure 4.2: The set of partial order relations justifying the observable behavior in question for both the candidates in Figure 4.1.

Consider another example in Figure 4.3. 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(orange box).

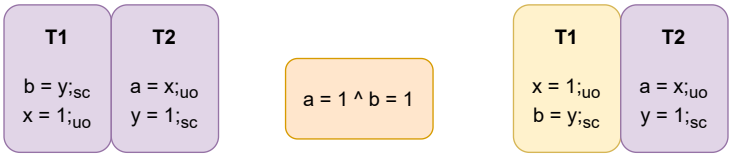

Figure 4.3: Second example for reordering with candidates of the original program and its reordered counterpart.

Figure 4.4 has two sets of relations. The first justifies that such an outcome is not possible for the original program candidate due to Axiom 1 applied to the read $a = x;_{uo}$ and the write $x = 1;_{uo}$ that it reads from. While the second justifies that this outcome is possible for the reordered program. Note that we cannot infer in the reordered candidate the set of relations for any candidate execution to have $a = x;_{uo} \xrightarrow[hb]{} x = 1;_{uo}$.

[1]In practice, this can be due to read speculation at the hardware level or due to reordering the two reads as an optimization that we show here.

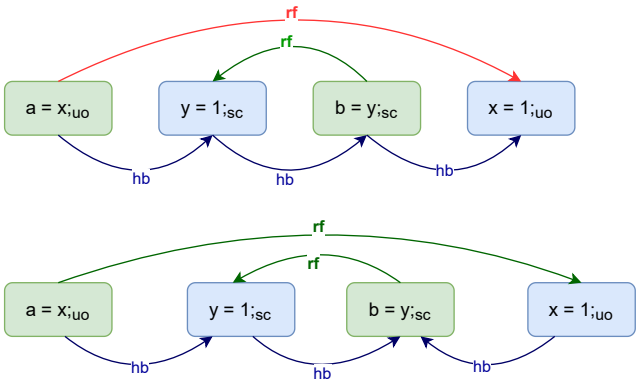

Figure 4.4: The set of partial order relations justifying the observable behavior in question for both the candidates in Figure 4.3.

The above two examples show that we have to be careful while reordering two events in the same thread. Naively, for each observable behavior, one must check all possible candidate executions and assert whether it is possible or not. It may also be 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.

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 feasible 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 transformations are automated in contrast to being done manually.

Our solution to this is to construct a proof at the candidate level, which can expose for Candidate Executions of the original program and the transformed one 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's set of Candidate Executions.

Assumption For this thesis, we only consider reordering among read/write events. We make the following assumptions for every program we consider:

1. All events are tear-free - Events that tear introduce many non-trivial behaviors, which we do not address in this thesis. We show a few examples of this in Chapter 6.

2. No synchronize events exist - Having them in programs would need an operational understanding of *wait/notify* events.

3. No Read-Modify-Write events exist - As we only consider reordering among read/write events.

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 $e \xrightarrow[ao]{} d$ becomes:

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

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

4.3 Some Useful Definitions

Before we go about proving when reordering is valid, we define certain helper definitions for it[2].

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 a direct* $\xrightarrow[ao]{}$ *relation among them; i.e those relations that are not derived through transitive property of* $\xrightarrow[hb]{}$.

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

[2]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.

Definition 4.3.2. *Direct happens-before relation (dir) We define* dir *to take an ordered pair of events* (e, d) *such that* $e \xrightarrow[hb]{} d$ *and gives a boolean value to indicate whether this relation is direct, which can be formally stated as follows:*

$$\nexists k \text{ s.t. } e \xrightarrow[hb]{} k \wedge k \xrightarrow[hb]{} d.$$

We can infer some useful things using dir *based on some information on events* e *and* d[3]*.*

1. *If* $e : uo$*, then* $dir(e, d) \Rightarrow cons(e, d)$.
2. *If* $d : uo$*, then* $dir(e, d) \Rightarrow cons(e, d)$.
3. *If* $e : sc \wedge e \in R$*, then* $dir(e, d) \Rightarrow cons(e, d)$.
4. *If* $e : sc \wedge e \in W$*, then* $dir(e, d) \Rightarrow cons(e, d) \vee e \xrightarrow[sw]{} d$.
5. *If* $d : sc \wedge d \in W$*, then* $dir(e, d) \Rightarrow cons(e, d)$.
6. *If* $d : sc \wedge e \in R$*, then* $dir(e, d) \Rightarrow cons(e, d) \vee e \xrightarrow[sw]{} d$.

.

Definition 4.3.3. *Reorderable Pair (Reord)*

We define a boolean function Reord *that takes two ordered pair of events* (e, d) *such that* $e \xrightarrow[ao]{} d$ *and gives a boolean value indicating if they are a reorderable pair*[4]*.*

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

[3]They can be proved trivially using definitions of $\xrightarrow[hb]{}$, $\xrightarrow[sw]{}$ and $\xrightarrow[ao]{}$

[4]We later prove that this exact definition defines when a pair of two consecutive events are reorderable.

4.4 Useful Lemmas

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

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

If

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

then,

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

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

Proof. We have the following to be true:

$$cons(e,d) \ \wedge \ e \xrightarrow[ao]{} d. \tag{1}$$

From 1 and by Definition of $\xrightarrow[hb]{}$, we can infer:

$$\nexists k \text{ s.t. } e \xrightarrow[hb]{} k \wedge k \xrightarrow[hb]{} d.$$

from which we can conclude

$$dir(e,d).$$

Now from

$$(d\colon uo) \ \vee \ (d\colon sc \ \wedge \ d\in W).$$

we can infer using Def 4.3.2 that for any event k

$$dir(k,d) \Rightarrow cons(k,d). \tag{2}$$

Because $\xrightarrow[ao]{}$ is a total order, $k = e$ will be the only solution satisfying 2. This means for any other $k \neq e$, we have $\neg dir(k,d)$, from which we can conclude that

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

Figure 4.5 summarizes the intuition behind the proof. It showcases the direct happens-before relations that can come to d.

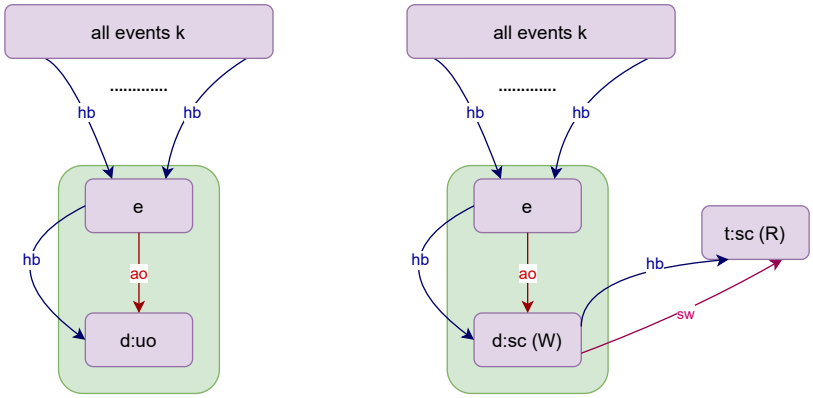

Figure 4.5: Two cases of lemma 1 proof.

□

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 the 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. We have the following to be true:

$$cons(e,d) \ \wedge e \xrightarrow[ao]{} d. \tag{1}$$

From 1 and by Definition of $\xrightarrow[hb]{}$, we can infer:

$$\nexists k \text{ s.t. } e \xrightarrow[hb]{} k \wedge k \xrightarrow[hb]{} d.$$

from which we can conclude

$$dir(e, d).$$

Now from

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

we can infer from Def 4.3.2 that for any event k

$$dir(e, k) \Rightarrow cons(e, k). \tag{2}$$

Because $\overrightarrow{ao}$ is a total order, $k = d$ will be the only solution satisfying 2. This means for any other $k \neq e$, we have $\neg dir(k, d)$, from which we can conclude that

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

Figure 4.6 summarizes the intuition behind both cases: It showcases the direct happens-before relations that can come to e.

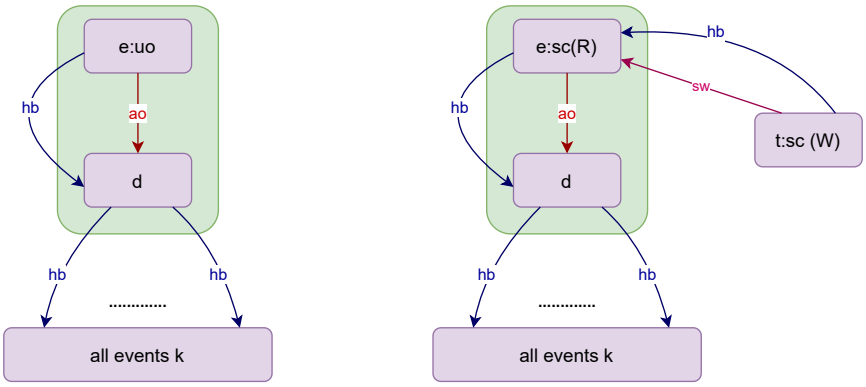

Figure 4.6: Two cases of lemma 2 proof.

□

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 a strict 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 $e \overrightarrow{hb} d$, do other *happens-before* relations remain intact?
2. Apart from $d \overrightarrow{hb} e$, 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 $\overrightarrow{hb}$ 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.

1. $k \overrightarrow{hb} e$.
2. $e \overrightarrow{hb} k$.
3. $d \overrightarrow{hb} k$.
4. $k \overrightarrow{hb} d$.

Firstly, note that the relations of the form (2) come through either a $\overrightarrow{sw}$ relation with e or relations through event d, i.e. of the form (3)). 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 (4) wherein relations derived through (1) 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.

$$K_e = \{k \mid k \overrightarrow{hb} e\}.$$
$$K_d = \{k \mid d \overrightarrow{hb} k\}.$$

Figure 4.7 below pictorially shows this.

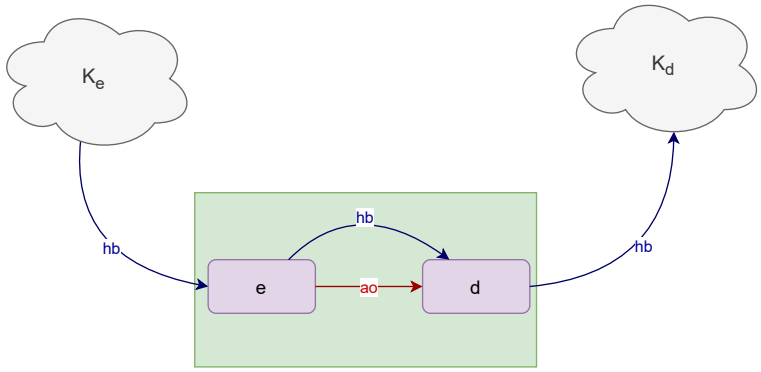

Figure 4.7: For any Candidate Execution of C, the set K_e and K_d and its relation with events e, d.

change the above figure to represent sets as clouds.

Consider two events $p1 \in K_e$ and $p2 \in K_d$ (When e is the first event or d is the last event in an agent event list, 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 shown in Figure 4.8

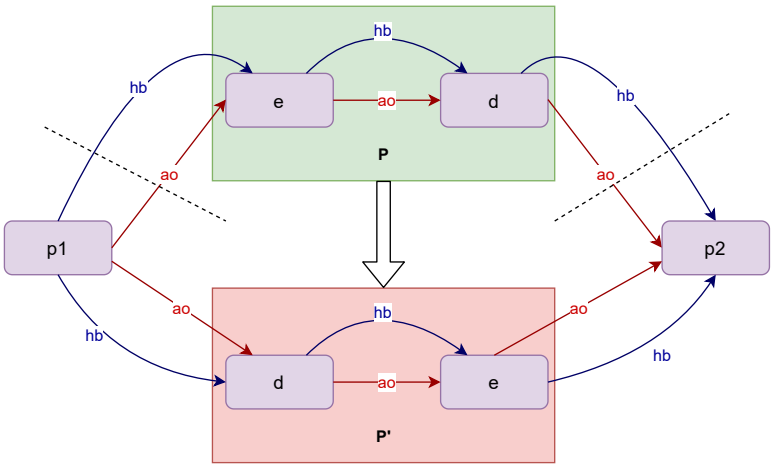

Figure 4.8: The *direct happens-before* relation that change while reordering events e and d.

Figure 4.8 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'

1. $p1 \xrightarrow[hb]{} e$.
2. $e \xrightarrow[hb]{} k$.
3. $d \xrightarrow[hb]{} p2$.
4. $k \xrightarrow[hb]{} d$.

After reordering, we have (1) and (3) preserved due to transitivity

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

(2) and (4) may not be preserved due to $d \xrightarrow[sw]{} k$ or $k \xrightarrow[sw]{} d$. If we can "pivot" the set K_e to $p1$ and K_d to $p2$, it would ensure that our other two intended relations also

remain preserved after reordering by transitivity. To state formally, we have a valid pair of pivots $< p1, p2 >$ when the following two conditions hold

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

Applying Lemma 1 for events $p1, e$ and Lemma 2 for events $p2, d$ respectively, we have for C, the following condition where $< p1, p2 >$ is a valid pivot pair.

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

Figure 4.9 summarizes the cases where we have a valid pair of pivots[5] $< p1, p2 >$. THe first cell describes the purpose of the table (here it is valid pair of pivots). The column represents the cases for the types of events e, d in that order. The row represents the cases of e, d being a read/write in that order. Note that all further tables summarizing our results will follow this format.

<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 for case where $e \in R \wedge e : sc \wedge d \in W \wedge d : uo$ (Figure 4.10 and 4.11) where we do not have a valid pair of pivots; particularly because $p1$ is not a valid pivot. Note that in this example, $K_e = K_{e1} \cup K_{e2} \cup \{p1\} \cup \{p_x\}$

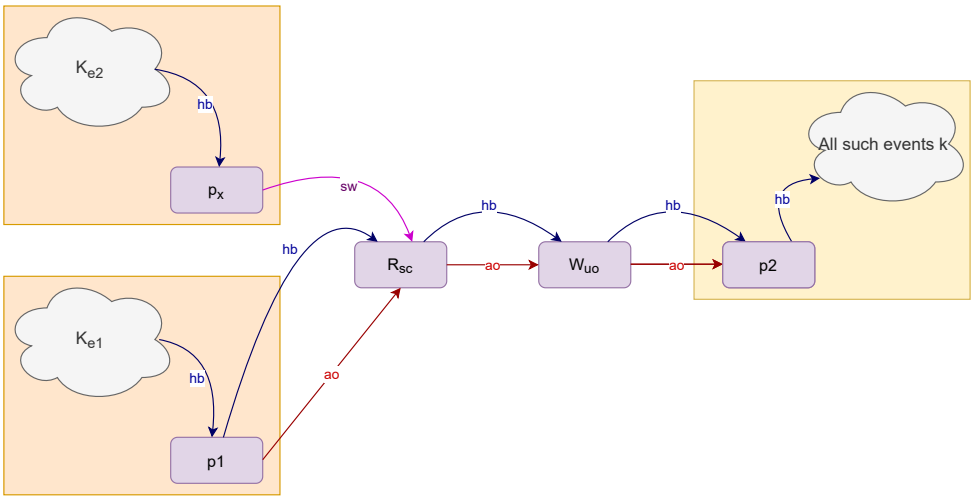

Figure 4.10: A Candidate Execution where p1 is not a valid pivot.

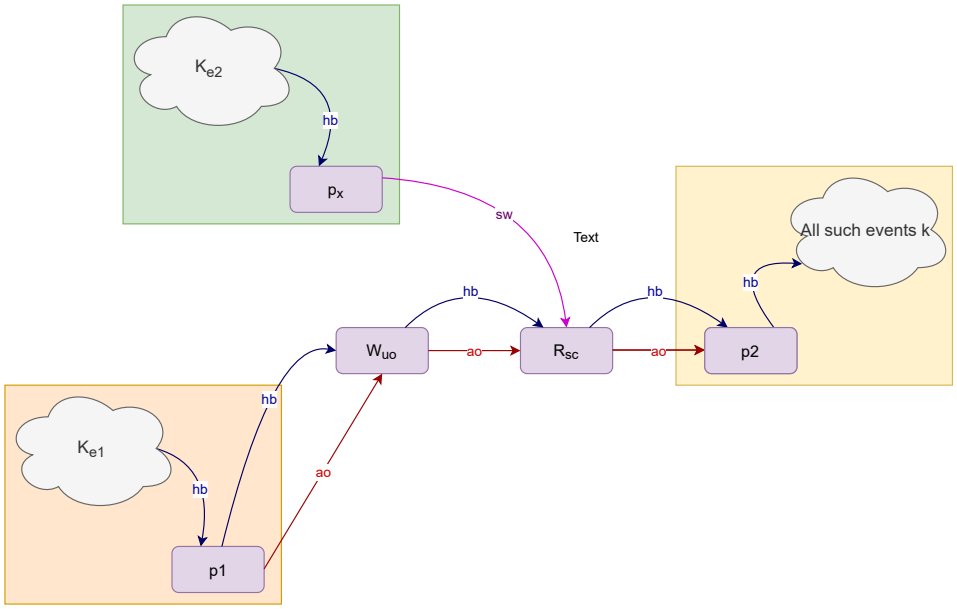

Figure 4.11: The resultant Candidate Execution after reordering, exposing the relations with p_x, K_{e2} and d that are lost

[5]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

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 with d. Thus, although we are able to preserve relations that existed in any $CandidateExecution$ of C, we also in the process, may introduce new ones in $CandidateExecutions$ of C'. Figures 4.12 and 4.13 shows pictorially an example of a set of relations that may exist in a Candidate Execution of C for the case above:

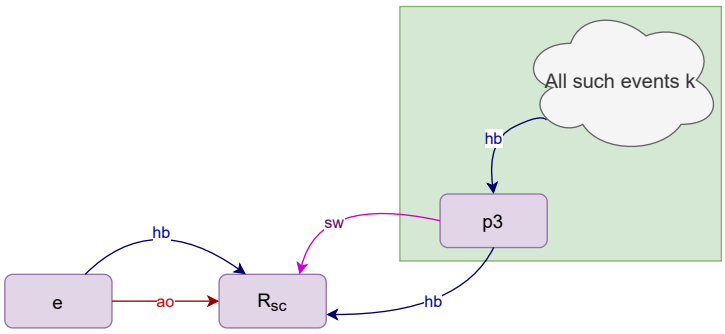

Figure 4.12: A Candidate Execution where d read having access mode *sc*.

that for any possible Candidate Execution after reordering, the set of happens-before relations apart from that between e and d remain the same.

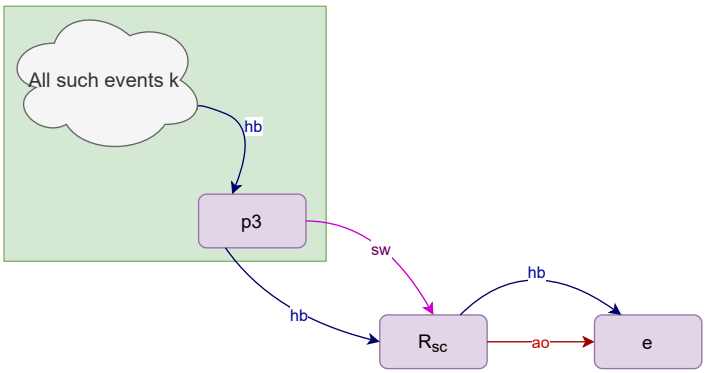

Figure 4.13: The Candidate Execution after reordering e and d, exposing the new relations established with e, $p3$ and set k

Figure 4.14 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 new *happens-before* relations that could be introduced on having valid pair of pivots.

For these cases, we investigate whether the new relations introduce new observable behaviors.

3. Presence of cycles? Before we go into analyzing whether new relations introduce observable behaviors, 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.
3. At least 3 events will be involved in the cycle.

The first part is straightforward as we only do reordering on Candidate Executions 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 third event k, may be either from the set K_e, K_d or a new relation that is formed.

Figure 4.15 shows that k cannot belong to either of the sets, as their relations with e and d will not result in a cycle.

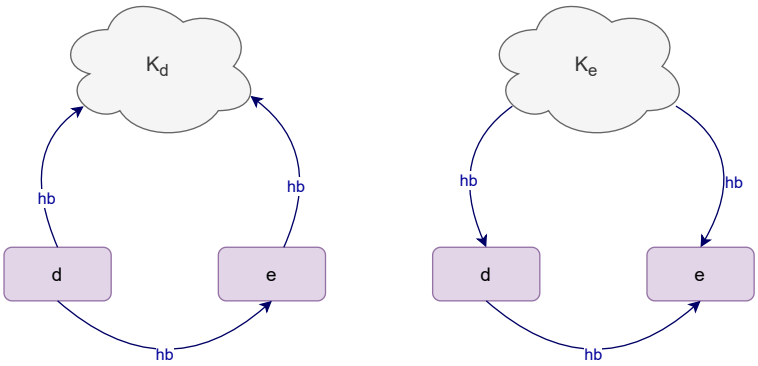

Figure 4.15: If event k is from K_e or K_d, there cannot be happens-before cycles.

For cases where $k \xrightarrow[hb]{} e$ is in the set of new relations, note that by Lemma 4.5

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

For cases where $d \xrightarrow[hb]{} k$ is the set of new relations, by Lemma 4.6

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

So for both these cases also, a cycle with $d \xrightarrow[hb]{} e$ cannot exist. Figure 4.16 shows pictorially this fact.

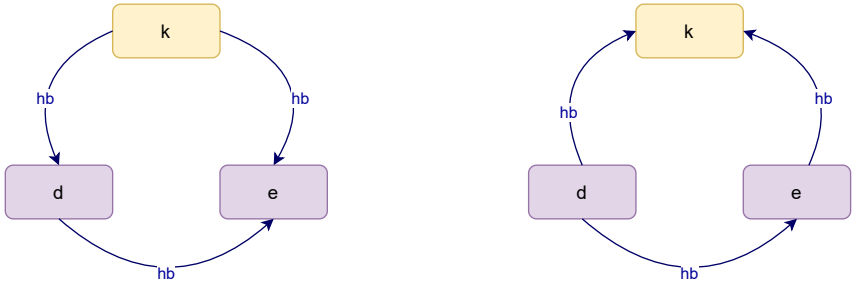

Figure 4.16: Sets of relations $k \xrightarrow[hb]{} e$ or $d \xrightarrow[hb]{} k$ individually do not create any cycles.

For the one case where we have two new sets of relations formed, i.e $d \xrightarrow[hb]{} k$ and $k \xrightarrow[hb]{} e$, we could have a case where k is a common event for both sets. But, by Lemma 1, we also have $k \xrightarrow[hb]{} d$ and by Lemma 2, $e \xrightarrow[hb]{} k$. Thus, we have a cycle. Figure 4.17 below shows this pictorially.

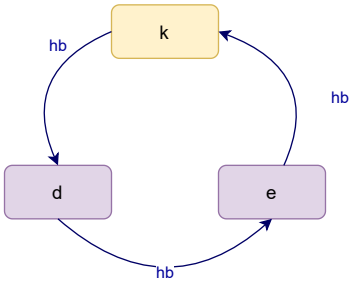

Figure 4.17: Relations $k \xrightarrow[hb]{} e$ and $d \xrightarrow[hb]{} k$ creates a cycle.

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

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

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

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

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

So, the following is also a cycle

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

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

In similar lines for the cases where the set of new relations are of the form $d \xrightarrow[hb]{} k$, 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

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

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.

Figure 4.18 shows the cases where new relations have no happens-before cycles.

hb-cycle	R-R	R-W	W-R	W-W
uo-uo	N	N	N	N
uo-sc	N	N	N	N
sc-uo	N	N	N	N
sc-sc	N	N	Y	N

Figure 4.18: Table summarizing *happens-before* cycles that may be introduced after reordering candidates with valid pivots.

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 Table 4.9:

1. $e \in R \ \wedge \ d \in R \ \wedge e : uo \ \wedge d : uo.$
2. $e \in R \ \wedge \ d \in R \ \wedge e : uo \ \wedge d : sc.$
3. $e \in R \ \wedge \ d \in W \ \wedge e : uo \ \wedge d : uo.$
4. $e \in W \ \wedge \ d \in R \ \wedge e : uo \ \wedge d : uo.$
5. $e \in W \ \wedge \ d \in R \ \wedge e : uo \ \wedge d : sc.$
6. $e \in W \ \wedge \ d \in R \ \wedge e : sc \ \wedge d : uo.$
7. $e \in W \ \wedge \ d \in W \ \wedge e : uo \ \wedge d : uo.$
8. $e \in W \ \wedge \ d \in W \ \wedge e : sc \ \wedge d : uo.$

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

- (1) and (2) 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($\Re$) between the two read events.
- (3) 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 is overlapping or equal.
- (4), (5) and (6) can fit in the pattern of Axioms 1 and 3, if e and d have overlapping/equal 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 behaviors.
- (7) and (8) can fit in the pattern of Axioms 1 and 3, if they have overlapping/equal 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.

The 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:

1. $e : uo \ \wedge \ e \in R \ \wedge \ k \xrightarrow[hb]{} e.$
2. $e : uo \ \wedge \ e \in W \ \wedge \ k \xrightarrow[hb]{} e.$
3. $d : uo \ \wedge \ d \in R \ \wedge \ d \xrightarrow[hb]{} k.$
4. $d : uo \ \wedge \ d \in W \ \wedge \ d \xrightarrow[hb]{} k.$

Figure 4.19 shows an exhaustive breakdown of sub-cases for case (1), varying based on the nature of event k.

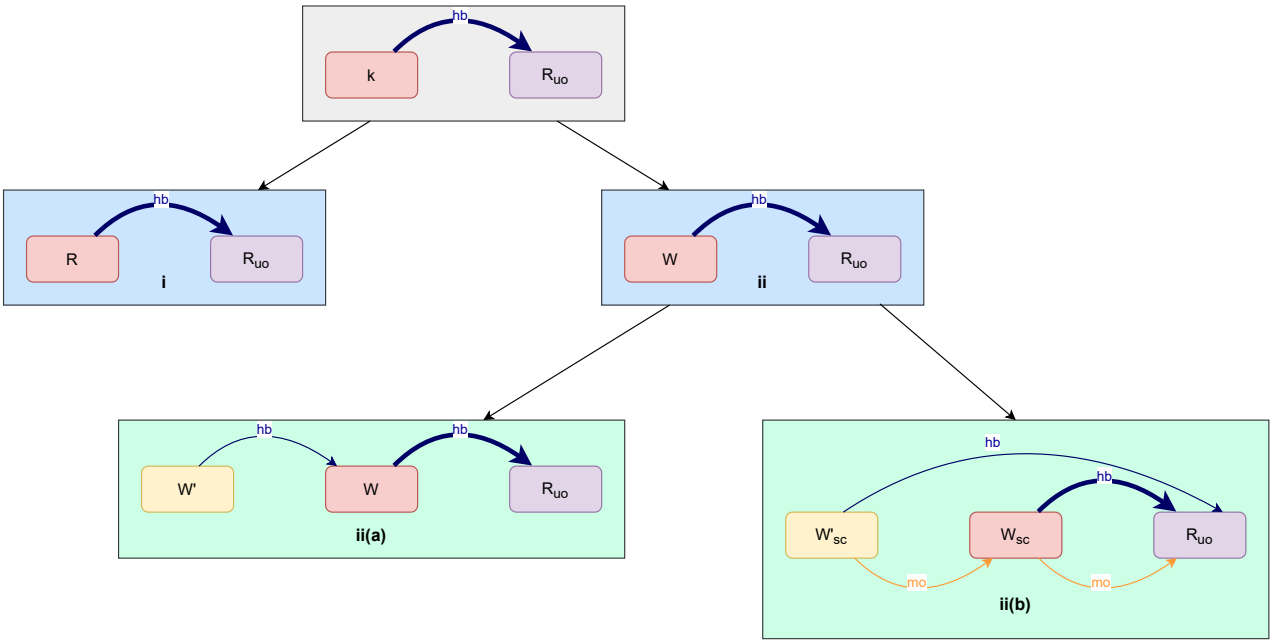

Figure 4.19: The impact of new relation $k \xrightarrow[hb]{} e$ where $e \in R \wedge e : uo$ on observable behaviors.

For case (1) we can observe the following from Figure 4.19

- 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.

Thus, for case (1), we can conclude that the new relations do not introduce any new observable behaviors, rather they only restrict possible behaviors.

Figure 4.20 shows an exhaustive breakdown of sub-cases for case (2), varying based on the nature of event k.

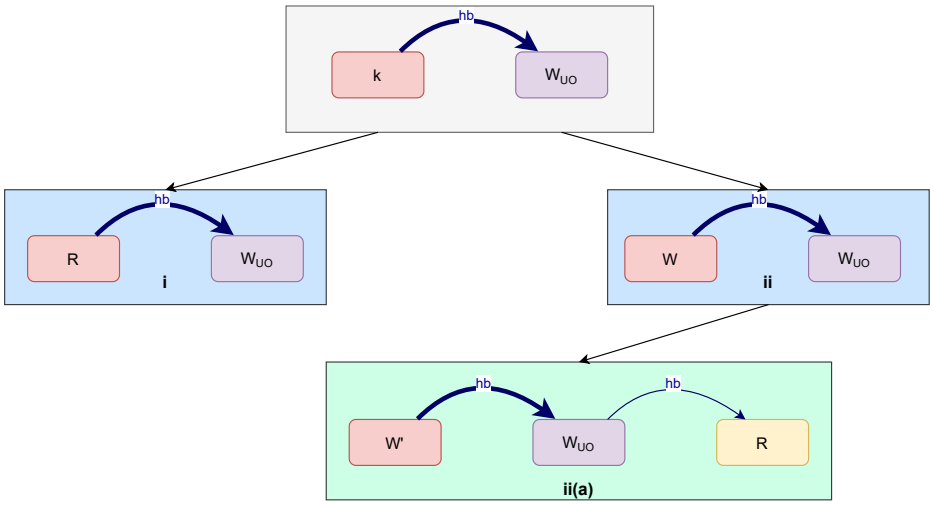

Figure 4.20: The impact of new relation $k \xrightarrow[hb]{} e$ where $e \in W \wedge e : uo$ on observable behaviors.

For case (2) we can observe the following from the Figure 4.20

- 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 (ii(a)) restricts some read from reading parts of k due to the write e.

Thus, for the case (2), we can conclude that the new relations do not introduce any new observable behaviors, rather they only restrict possible behaviors.

Figure 4.21 shows an exhaustive breakdown of sub-cases for case (3), varying based on the nature of event k.

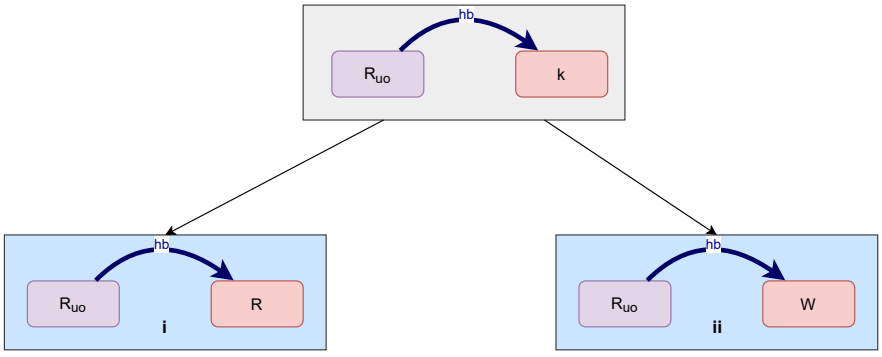

Figure 4.21: The impact of new relation $d \xrightarrow[hb]{} k$ where $d \in R \wedge d : uo$ on observable behaviors.

For case (3), we can observe the following from Figure 4.21

- 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.

Thus, for the case (3), we can conclude that the new relations do not introduce any new observable behaviors, rather they only restrict possible behaviors.

Figure 4.22 shows an exhaustive breakdown of sub-cases for case (4), varying based on the nature of event k.

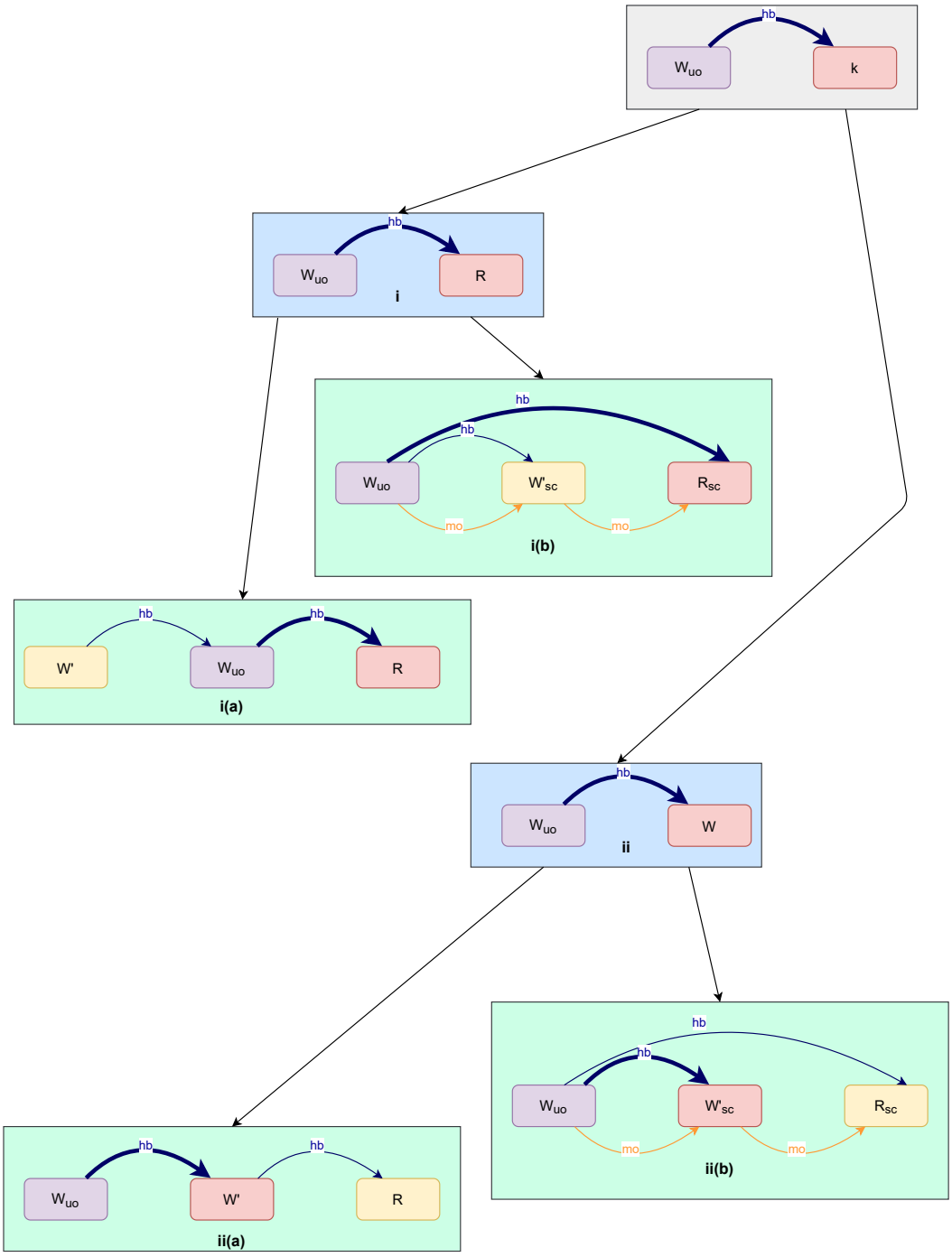

Figure 4.22: The impact of new relation $d \xrightarrow[hb]{} k$ where $d \in W \wedge d : uo$ on observable behaviors.

For case (d) we can observe the following from Figure 4.22

- 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,

Thus, for the case (4), we can conclude that the new relations do not introduce any new observable behaviors, rather they only restrict possible behaviors.

The above case wise analysis show us that any new relation (apart from $d \xrightarrow[hb]{} e$), matching the patterns of the axioms, only restrict possible observable behaviors ($\xrightarrow[rf]{}$ relations). Thus, we can conclude that no new observable behavior is introduced due to the new set of $\xrightarrow[hb]{}$ relations.

In summary, Figure 4.23 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 4.23: 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 valid Candidate Executions. Consider two events e and d such that* $\neg cons(e,d)$ *is true in C and* $e \overrightarrow{_{ao}} d$*. Consider another Candidate C' resulting after reordering e and d in C. If*

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

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

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

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

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

What we want after reordering is C', with

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

whose observable behaviors is subset of C.

Without loss of generality, we can choose to first reorder k and d. With $Reord(k, d)$, we can, from Theorem 4.1, reorder them, giving us candidate C'' with

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

whose observable behaviors is subset of C.

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

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

whose observable behaviors is subset of C''.

Now lastly, we need to reorder e and k for which we need $Reord(e, k)$ to hold, thus by Theorem 4.1, giving us our final candidate C' with

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

whose observable behaviors is subset of C'''.

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

2. Inductive Case $n > 1$ Assume the above corollary holds for $n = t$, meaning the observable behaviors of candidate C'_t is a subset of C_t. We need to show that for $n = t + 1$, the corollary still holds, for this note firstly that, we have the following ordering relations:

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

Without loss of generality, we can first reorder k_{t+1} and d. To do this, we need $Reord(k_{t+1}, d)$ to hold, thus by Theorem 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 Theorem 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. □

The next two corollaries will help us prove the validity of reordering when loops and conditionals are involved.

Corollary 4.1.2. *Consider a Candidate C of a program and its valid Candidate Executions. 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 difference 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 transitivity of subset relations, the corollary holds. □

Corollary 4.1.3. *Consider a Candidate C of a program and its valid Candidate Executions. 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 difference 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 transitivity of subset relations, the corollary holds. □

For cases where reordering is not safe to do, we also show counter examples of programs where new observable behaviors are introduced. This additionally helps gain intuition about the proof given. Note that we do not show examples for cases where there exists data dependancies between e and d itself. We show counterexamples where $\overrightarrow{hb}$ relations lost (those across agents specifically), could introduce new observable behaviors. These examples are in Appendix A.1.

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 reorder two events at the program level, we must also address the involvement of conditionals and loops that may be between these two events. But locally asserting why a particular reordering is valid is non-trivial to understand. The compiler does global level optimizations that involve several local transformations that may include reordering. In addition, the previous proofs show us that we mainly are interested in identifying whether a reordering introduces new observables, not with why such a reordering would be effective for performance.

Hence, the way we approach this is to not have any assumptions as to why the compiler chooses to do a particular reordering. Instead, we only check if the reordered program can have its observable behaviors as a subset of the original. This ensures that we do not concern ourselves with the algorithm behind the compiler optimization. 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.

We first define two properties that relate conditionals to Candidates of a program. We then use these properties to state a lemma which will be useful to prove the corollary that follows which establishes the condition under which reordering in programs with conditionals is safe.

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 *and consider* k *to be a representative event outside the conditional branch. Then* $b1 \in B1$ *if and only if:*

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

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, whether its an "if-then"(1-branch) clause or "if-then-else"(2-branch) clause. 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*. Then* $b1 \in B1 \ \wedge \ b2 \in B2$ *if and only if:*

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

There cannot exist any candidate of the program such that events from both sets can be part of it[7]*.*

Figure 4.24 summarizes the two forms of conditionals we can have in any program.

[7]Note that here we consider every statement in the program unique. So a program like "if (c) then $x = 1$; else $x = 1$;", both the writes to x are unique.

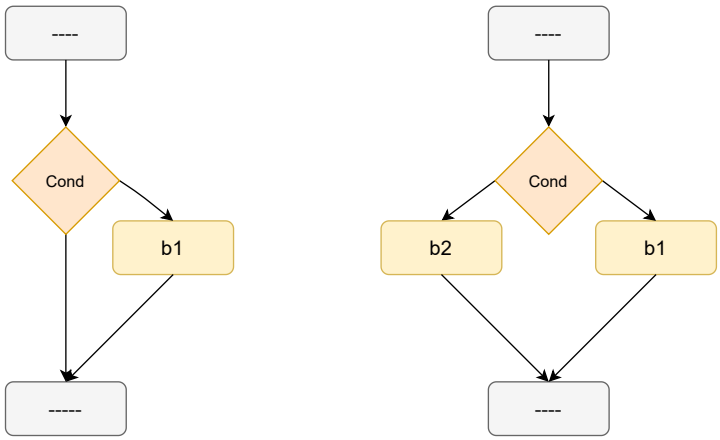

Figure 4.24: Two forms of conditionals.

We use the above two properties to state the following lemma

Lemma 3. *Reordering an statement e inside a conditional to outside a conditional violates Property 1 and Property 2*

Proof. The proof is trivial. By removing a statement outside of a conditional branch, we can get a candidate of a program that would violate both properties. □

The above proof also lets us infer that on reordering an event outside a conditional, there are Candidates that 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:*

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

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

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

Proof. 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 conditional, then by Property 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 conditional 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, without 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 4.25:

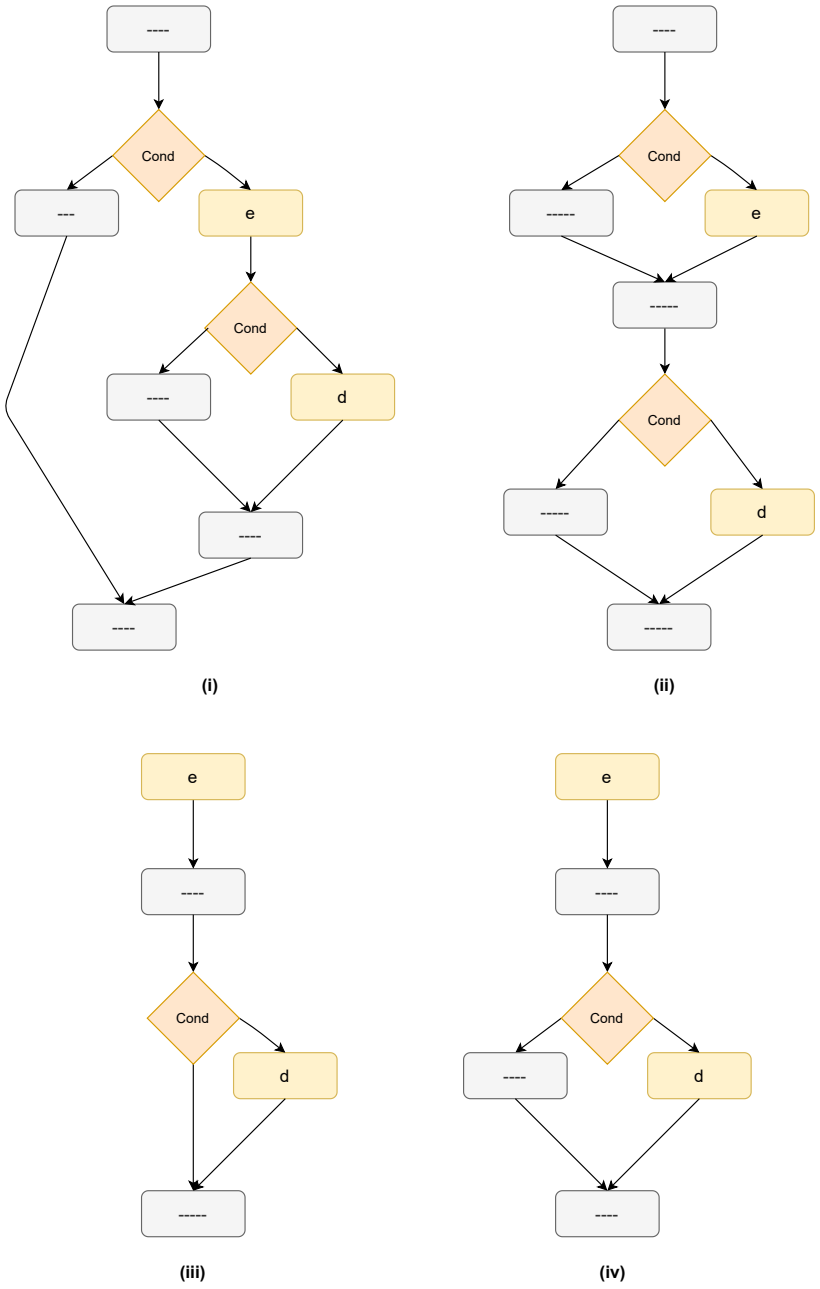

Figure 4.25: Four base cases where e or d are part of some conditional branch.

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. Thus, we have a new observable behavior[8].

Hence, by contradiction, the second condition must hold.

The first condition holds trivially as it corresponds to Corollary 4.1.1 for each candidate $C_{i\in[1,n]}$. By property of union of sets, we can infer that the set of Observable Behaviors of P' is a subset of that of P.

□

In addition to the above proof, we also show certain counter examples where reordering in the presence of conditionals may not be safe to do. This facilitates better understanding of the proof and its arguments. These counter examples are with respect to reordering of writes. These examples are in Appendix A.2.

4.6.2 Addressing Programs with Loops

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

There will be one Candidate for each possible count of loop iteration. 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 to be an event within the loop of the program which in a candidate signifies the i^{th} iteration of the corresponding statement.

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 $C^1, C^2, ...$ 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 $C'^1, C'^2, ...$ after reordering e and d in*

[8]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 different reads-from relations. This could be done as a whole by addressing redundancy introduction optimization. This is not within the scope of the thesis.

P and its corresponding program P'. If the following three conditions hold:

$$Reord(e, d) \tag{4.1}$$

$$\forall C^i \in P,\ \forall j \in [1, i], \forall k\ s.t.\ e^j \xrightarrow[ao]{} k \wedge k \xrightarrow[ao]{} d^j\ .\ Reord(e^j, k) \wedge Reord(k, d^j) \tag{4.2}$$

$$\nexists C^i \in P\ s.t.\ \forall j \leq i,\ (e^j \in C\ \wedge\ d^j \notin C)\ \vee\ (e^j \notin C\ \wedge\ d^j \in C) \tag{4.3}$$

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

Proof. The proof for this is fairly straightforward. Condition 4.1 corresponds to Theorem 4.1. Condition 4.2 and 4.3 correspond to Corollary 4.1.1 and 4.1.4 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. These conditions precisely define this requirement[9].

□

Reordering Accross Loops What is not so obvious is the case when events are reordered out of the loop. The problem is that we cannot use parent Candidate to generate resultant 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. This is because, such a transformation implies introduction/elimination of events in a Candidate. We will show in the next chapter that we can in fact 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

[9]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 straightforward flow-analysis by the compiler. Additionally, Condition 4.3 can always be checked beforehand as it corresponds to checking whether events e and d belong in different conditional branches.

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 elimination under the ECMAScript Memory Model.

Chapter 5

Elimination

This chapter investigates the validity of elimination under the ECMAScript memory model. We first start by explaining using examples why elimination is not safe in the relaxed memory context. We then prove the validity of read elimination followed by that for write at the Candidate level. Similar to reordering, we further move to prove elimination at the program level (still abstracted to a set of shared memory events) involving conditionals and loops. We lastly prove a subset of loop-invariant code-motion using our results from proving elimination and reordering at the program level.

5.1 Introduction

Many programs fall victim to having lots of redundant code. Right from simple redundant writes to outright dead code, removing them plays a major role in performance of our programs. Such redundancy could also be possible due to different phases of optimization the compiler performs, which leaves certain residual code in each phase. Examples of such optimizations are common-sub-expression elimination, copy propagation, partial redundancy elimination, register allocation, etc.

In a sequential setting the effect of removing such code is expected to be invisible to execution. However, as we saw for instruction reordering, in a concurrent setting, elimination may not be that straightforward.

Write Elimination Consider the first example in Figure 5.1 below of a Candidate (left) and the resultant candidate after eliminating a write (right). The orange box shows the observable behavior that we want to consider. Note that here, the outcome of read values of b and c are not relevant for our purpose. Hence they can have any value.

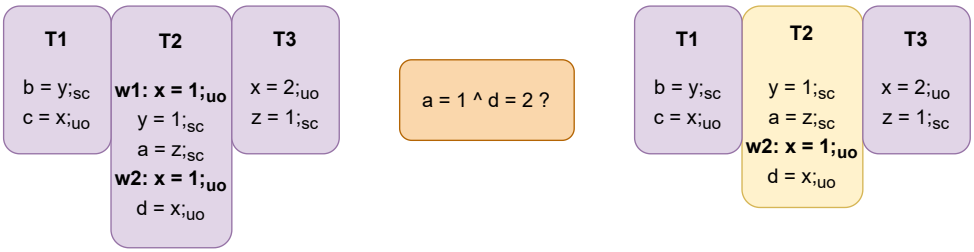

Figure 5.1: First example for elimination with candidates of the original program and its counterpart after elimination of a write.

Figure 5.2 explains the relations implied in a candidate execution based on Observable behaviors that disallows the behavior in question (top) for the original program but is valid for that after elimination (below).

The first set of relations is for the original program, where Axiom 1 prohibits the read d to have value of x as 2. For a to have value 1, it must come from the write $z = 1;_{sc}$. From this we can infer $\{z = 1;_{sc}\} \xrightarrow{hb} \{a = z;_{sc}\}$. From Def of agent order and happens-before, we can infer $\{x = 2;_{uo}\} \xrightarrow{hb} \{w2 : x = 1;_{uo}\} \xrightarrow{hb} \{d = x;_{uo}\}$. By Axiom 1, the read value of d cannot be 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). For a to have value 1, it must come from the write $z = 1;_{sc}$. From this we can infer $\{z = 1;_{sc}\} \xrightarrow{hb} \{a = z;_{sc}\}$. From Def of agent order and happens-before, we can infer $\{x = 2;_{uo}\} \xrightarrow{hb} \{d = x;_{uo}\}$. The above relations do not restrict d from having value 2.

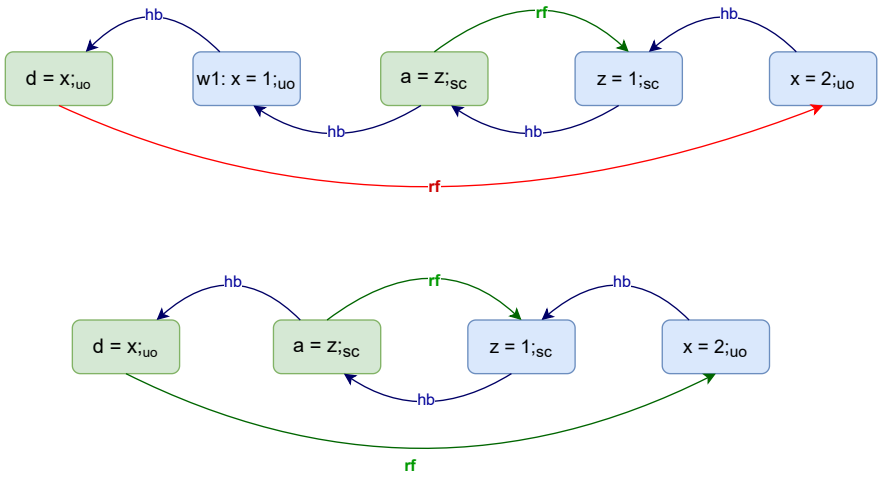

Figure 5.2: The set of partial order relations justifying the observable behavior in question for both the candidates in Figure 5.1.

In the same above example, if the compiler decides to eliminate $w1$, then the observable behavior $b = 1^c = 0$ will be allowed (we leave it as an exercise for the reader to verify). This is a classic example of where, a sequentially sane optimization by elimination is not possible, whatever the choice of write we choose to eliminate.

Read Elimination In the case of read elimination, we clearly would lose observable behaviors w.r.t. to the read eliminated. However, our focus is more on whether eliminating this read introduces new observable behaviors in the remaining program, meaning, can other reads be affected. Constructing such an example is non-trivial and involves implied memory order edges that are created between events. Figure 5.3 shows such an example where the Original Candidate (left) and the one after eliminating a read in $T2$ is shown. The orange box represents the behavior in question.

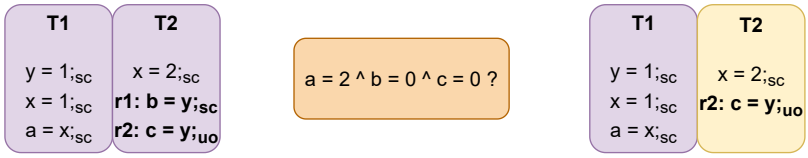

Figure 5.3: First example for elimination with candidates of the original program and its counterpart after elimination of a read.

Figure 5.4 explains the relations formed in a candidate execution that disallow

the behavior in question for the original program but justify it after elimination.

The first set of relations are for the original program, where the outcome in question is not allowed. For a to have value 2, it must come from the write $x = 2;_{sc}$. From this, we can imply $\{x = 2;_{sc}\} \xrightarrow{hb} \{a = x;_{sc}\}$. From Def of agent order, we can imply $\{x = 1;_{sc}\} \xrightarrow{hb} \{a = x;_{sc}\}$. From the above relations, we can imply the memory order $\{x = 1;_{sc}\} \xrightarrow{mo} \{x = 2;_{sc}\}$ (note that if the memory order was reversed, Axiom 3 would disallow the read value of a to be 2). By Def of memory order, we can now imply $\{y = 1;_{sc}\} \xrightarrow{mo} \{b = y;_{sc}\}$. We already have by Def of happens-before that $\{y = 0;_{init}\} \xrightarrow{hb} \{b = y;_{sc}\}$ and $\{y = 0;_{init}\} \xrightarrow{hb} \{y = 1;_{sc}\}$. The above relations, by Axiom 3 prohibits the read b to have value of y as 0. Thus b can only have the value 1. This in turn results in a happens-before relation $\{y = 1;_{sc}\} \xrightarrow{hb} \{b = y;_{sc}\}$, which then by Axiom 1 prohibits the read c to have value of y as 0.

The second set is for the modified program, where none of the axioms restrict such a behavior. We still retain the relations $\{x = 2;_{sc}\} \xrightarrow{hb} \{a = x;_{sc}\}$, $\{x = 1;_{sc}\} \xrightarrow{hb} \{a = x;_{sc}\}$ and $\{x = 1;_{sc}\} \xrightarrow{mo} \{x = 2;_{sc}\}$. But now, we have just $\{y = 1;_{sc}\} \xrightarrow{mo} \{c = y;_{uo}\}$, $\{y = 0;_{init}\} \xrightarrow{hb} \{c = y;_{uo}\}$ and $\{y = 0;_{init}\} \xrightarrow{hb} \{y = 1;_{sc}\}$. The above set of relations, does not restrict the read value of c to be 0. Thus, such an elimination is not sound for this example[1]..

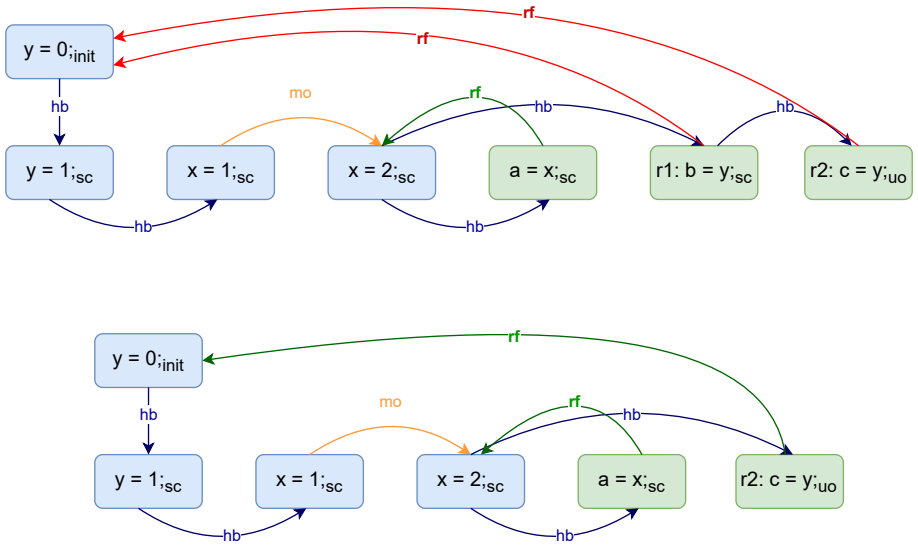

Figure 5.4: The set of partial order relations justifying the observable behavior in question for both the candidates in Figure 5.3.

[1]The avid reader may note that as long as the read value of a is 2, the value of c can never be

5.2 Approach

We consider the same set of assumptions for reordering here. Similar to reordering, our main objective is to ensure that the set of possible observable behaviors of a program, remain unchanged after elimination. In the case of write elimination, the property we try to prove remains the same. In the case of read elimination though, we would want the observable behaviors excluding the specific read eliminated to be a subset. For both cases, if preserving all behaviors is not possible, then we would want the set of observable behaviors after elimination at the very least to be a subset.

The main difference here is that elimination would remove certain happens-before relations, in contrast to having additional ones. From our point of view, we would want only the relations with the eliminated read/write to be removed after the transformation. The loss of these relations would certainly not have any new happens-before cycle introduced. However, we still have to check whether the removed relations result in some new behavior. We prove when it does not, by doing case-wise analysis on the type of relations eliminated.

5.3 Valid Elimination at the Candidate level

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. At the Candidate level, eliminating an event would imply removal of certain ordering relations at the Candidate Execution level. Given an event e belonging to a Candidate C, the candidate C' after eliminating event e from it would imply no relations of the form

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

and no relations of the form

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

exist in any Candidate Execution of C'.

0. But as we eliminate b, none of the axioms disallow the behavior in question.

5.3.1 Elimination of Reads

The following theorem establishes the condition when we can eliminate a read from a Candidate, while still ensuring that the result Candidate has observable behaviors as a subset.

Theorem 5.1. *Consider a candidate C of a program and its possible Candidate Executions where $\overrightarrow{hb}$ is a strict partial order. Consider an event e in C such that $e \in R$. Consider Candidate C' after eliminating the event e from C. If e has access mode uo, then the set of Observable behaviors of C' is a subset of C without the relation $e \overrightarrow{rf} w$ where w is some write event in both C and 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 implies removal $\overrightarrow{hb}$ relations, in contrast to introducing new ones. 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 derived through transitivity using relation with e, viz. using the following two relations:

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

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

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

We need to ensure the following relations hold after elimination.

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

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

$$\forall k_b \neq p_b \in K_b \ . \ k_b \overrightarrow{hb} p_b.$$
$$\forall k_a \neq p_a \in K_a \ . \ p_a \overrightarrow{hb} k_a.$$

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 $< p_b, p_a >$. Taking the conjunction of the above two constraints, we get $e : uo$ as the only possibility in which a valid pivot pair can exist.

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 possible $\xrightarrow[rf]{}$ relations other than those with e. We divide our argument into two parts, viz. the two types of relations removed:

1. $k \xrightarrow[hb]{} R_{uo}$.
2. $R_{uo} \xrightarrow[hb]{} k$.

Figure 5.5 shows an exhaustive breakdown of sub-cases for case (1), varying based on the nature of event k.

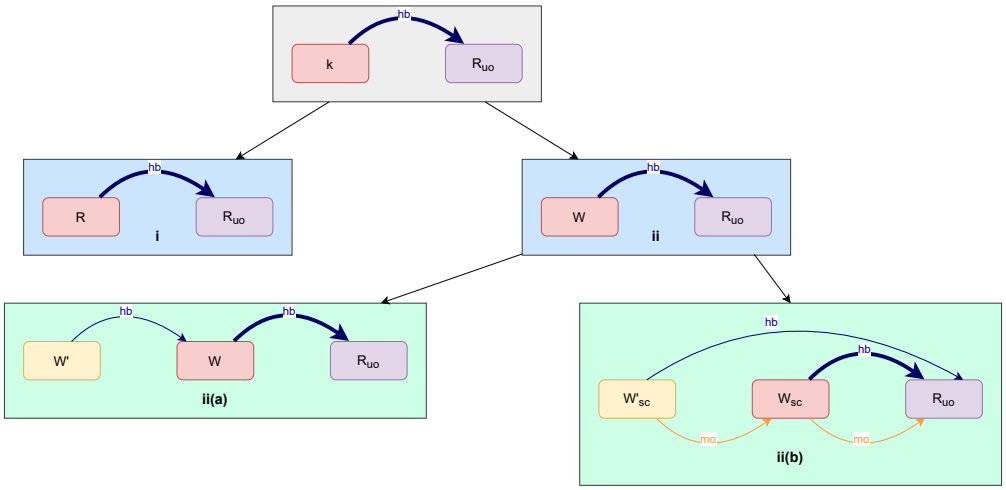

Figure 5.5: The impact of lost relation $k \xrightarrow[hb]{} R_{uo}$ on observable behaviors.

Observations:

- 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.

Thus, for the case (1), we can conclude that the removed relations do not introduce any new observable behaviors.

Figure 5.6 shows an exhaustive breakdown of sub-cases for case (2), varying based on the nature of event k.

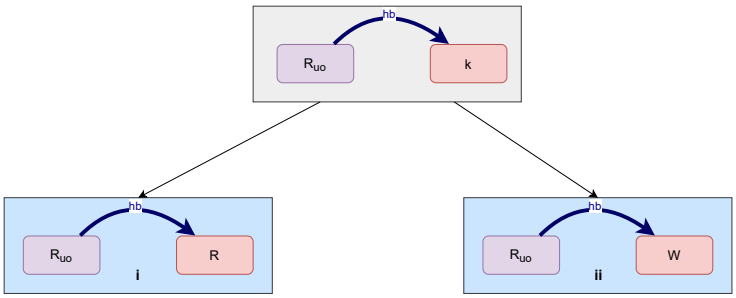

Figure 5.6: The impact of lost relation $R_{uo} \overrightarrow{hb} k$ on observable behaviors.

Observations:

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

Thus, for the case (2), we can conclude that the removed relations do not introduce any new observable behaviors.

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. □

5.3.2 Elimination of Writes

The following theorem establishes the condition when we can eliminate a write from a Candidate, given that we have another write consecutive to it.

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 any Candidate C' after eliminating the event e from C. If

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

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. We then identify relations lost due to elimination and a proof for when they do not introduce new observable behaviors.

Preserving Happens-before relations The relations we want to preserve are those that are derived 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 is 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.

We need both the above conditions to be satisfied to have a valid pivot pair $< p_b, p_a >$. Taking the conjunction of the above two constraints, we get $e : uo$ as the only possibility in which a valid pivot pair can exist.

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 $\overrightarrow{hb}$ relations are introduced, and because original candidate executions have $\overrightarrow{hb}$ 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:

1. $k \overrightarrow{hb} W_{uo}$.
2. $W_{uo} \overrightarrow{hb} k$.

Figure 5.7 shows an exhaustive breakdown of sub-cases for case (1), varying based on the nature of event k.

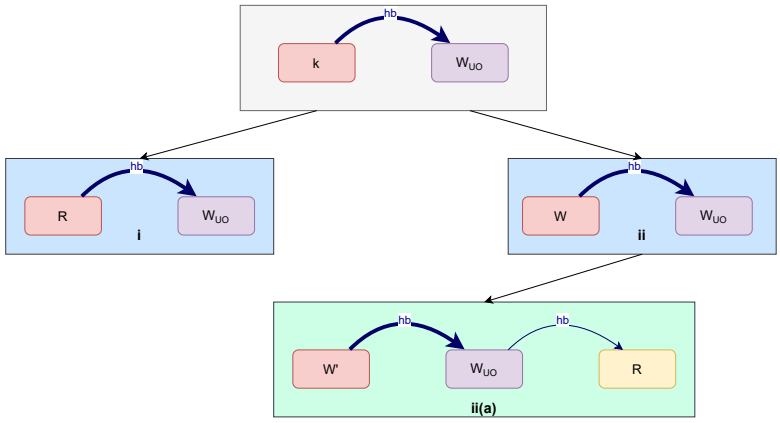

Figure 5.7: The impact of lost relation $k \overrightarrow{hb} W_{uo}$ on observable behaviors.

We can observe the following:

- (i) is a pattern from Axiom 1 that restricts the read R reading from e. This reads-from restriction will remain the same after elimination of e.
- (ii)(a) is a pattern from Axiom 1, forbidding R to read some bytes of W'. The set of byte-level restrictions will remain the same after elimination, if firstly we have the two relations

$$d \overrightarrow{hb} R \wedge W' \overrightarrow{hb} d.$$

Secondly, we need to ensure that after elimination, Axiom 1 now restricts the exact set of $\overrightarrow{rbf}$ relations with W' and R as before. By Lemma 2 and by Def of happens-before, the first condition holds. For the second, since R or W' can be arbitrary events with arbitrary ranges, we would need the ranges of e and d to be same (i.e $\Re(e) = \Re(d)$).

Thus, for the case (1), we can conclude that the removed relations do not introduce any new observable behaviors, while also requiring the constraint on the range of events e and d.

Figure 5.8 shows an exhaustive breakdown of sub-cases for case (2), varying based on the nature of event k.

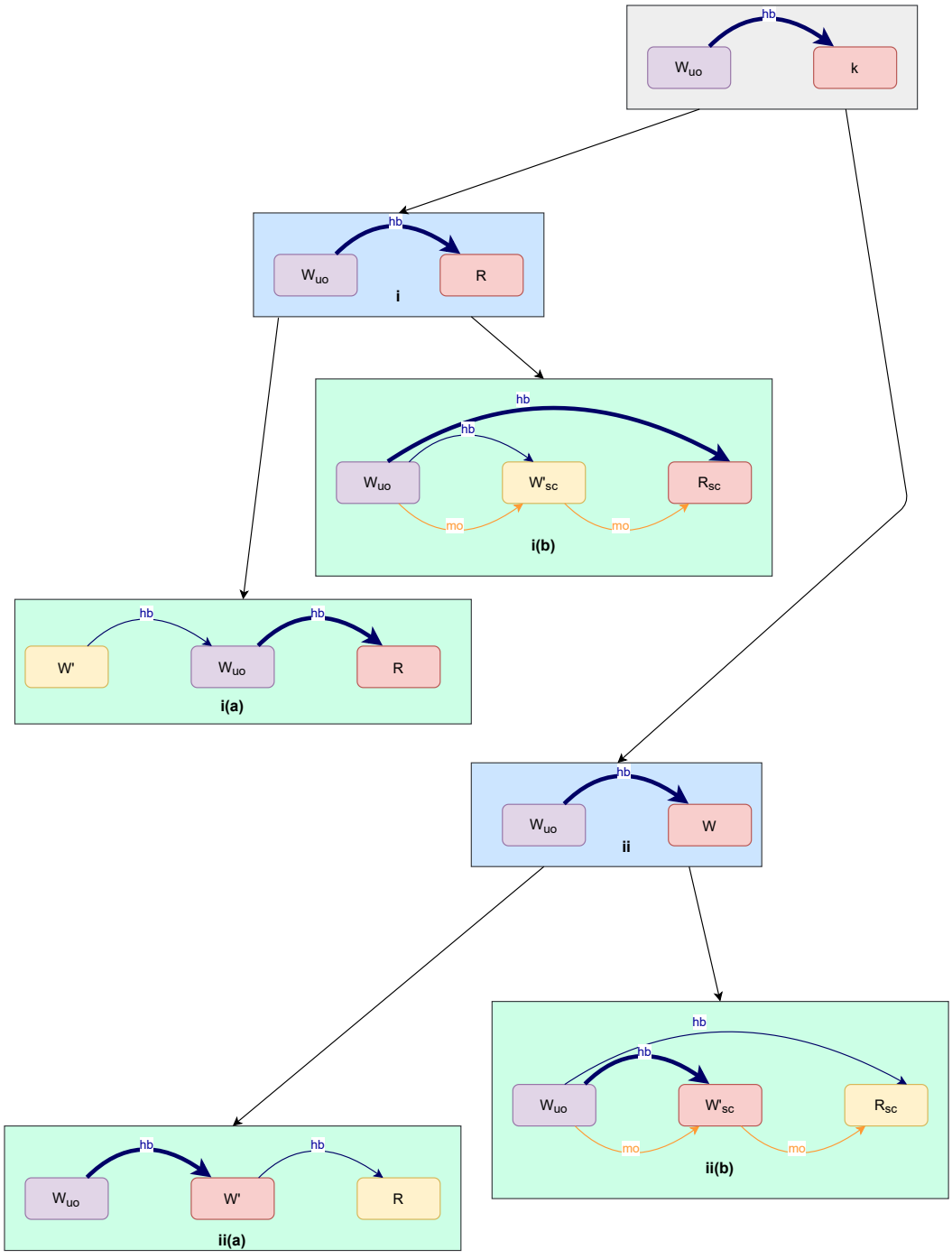

Figure 5.8: The impact of lost relation $W_{uo} \overrightarrow{hb} k$ on observable behaviors.

We make the following observations:

- (i)(a) has a similar argument to the case (1)'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 reads-from restriction will remain the same after e is eliminated.
- (ii)(a) is a pattern from Axiom 1, restricting R from reading W. This reads-from restriction will remain the same after after eliminating e.
- (ii)(b) is the same as (i)(b).

Thus, for the case (2), we can conclude that the removed relations do not introduce any new observable behaviors.

In all the above cases, we 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 unordered, the set of Observable Behaviors without the write is a subset of that with it present.

□

Theorem 5.2 establishes the condition when we can eliminate a write given that there exists another write with equal range consecutive to it. We now establish the condition when we can eliminate such a write without having a constraint on another consecutive write to be present. The following corollary along with its proof describes this exact scenario.

Corollary 5.2.1. *Consider a Candidate C of a program and its Candidate Executions which are valid. Consider two events e and d such that:*

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

Consider another Candidate C' after eliminating the event e from C. 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 an integer 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 4.1 and Def 4.3.1, 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 between e and d 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, without loss of generality, we can label all $n+1$ events k events as $k_1, k_2, ..., k_{n+1}$ that hold the following:

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

By Theorem 4.1 and Def 4.3.1, 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}.

□

5.4 From Candidates to Program

At the program level, similar to that of reordering, we first address programs with just conditional branches. We establish a conservative condition under which elimination of write in the presence of conditionals is safe, followed by justifying for that of reads.

We then move on to addressing programs with loops. Within this section, we first address elimination of a read in the presence of loops. We then state a corollary that establishes the condition under which write elimination is safe in the presence of loops. In both the above cases, we assume the eliminated event belongs within the loop.

Lastly, we establish the conditions under which loop invariant code motion is sound (4 corollaries with proof) using both our results from that of reordering and elimination of events.

5.4.1 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 establishes 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

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

$$\nexists C \ s.t. \ e \in C \wedge d \notin C. \tag{2}$$

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

Proof. From Condition 1 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 Condition 2 does not hold. Then we have

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

This would mean, for such a candidate C 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, Condition 2 must hold[2].

Figure 5.9 elicit the cases of conditionals that are allowed by the Condition 2 below.

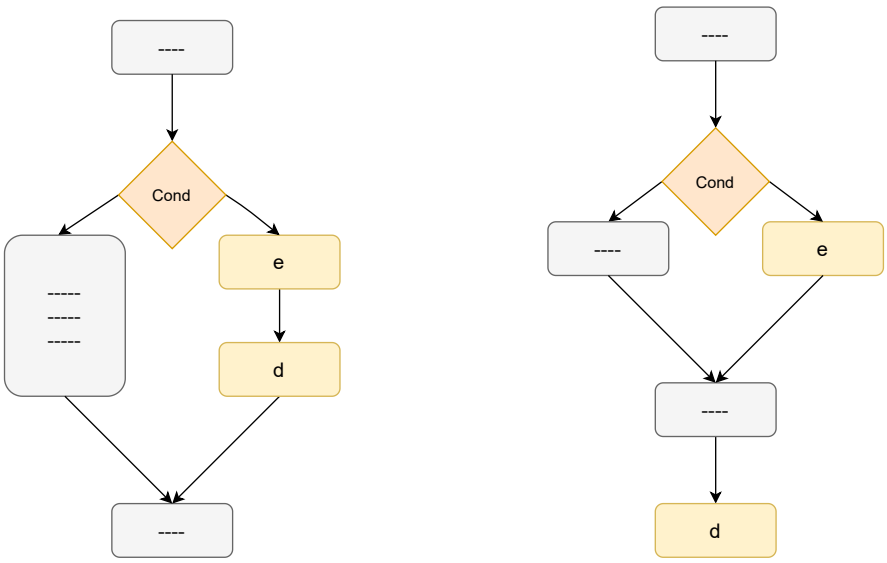

Figure 5.9: Cases of Conditionals where elimination of e is safe.

□

As far as read elimination goes, since we only need the information of the 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. When this is the case, the resultant code after

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

elimination depends on the intention of the compiler, which can be either of the following:

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

Since we place no assumption on why the compiler would do such elimination, it is not possible to ascertain the target code and their resultant Candidate and Candidate Executions that are intended. 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.

5.4.2 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 within the loop can be eliminated.

By Theorem 5.1, we only need the read to have the type *uo* and not be a conditional check 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 candidates 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 write 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

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

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 conditionals 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 candidates extends to program level by property of union of sets and subset relations[3].

□

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, reordering an event outside a loop would in principle involve eliminating the extra events from multiple iterations, when looked at from the Candidate level. Now that we have proven when elimination is valid under the relaxed memory model, we notice that 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 argument; the resultant program after transformation has observable behaviors as a subset of the original.

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

[3]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.

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, not being a conditional check. 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.

Proof. We first consider the programs with just one iteration of the loop; Candidates of the form C^1, having just e^1. We need to ensure that the resultant candidates $C^{1\prime}$ 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. Using Condition 5.7, we have from Corollary 4.1.2 that $C^{1\prime}$ 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. Using Condition 5.8, we have from Prop 1 that e is not part of any conditional branch. Thus, from Corollary 4.1.4, we can infer that the transformed program has observable behaviors a subset of the original.

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 $C^{2\prime}$ with just one such event e, such that:

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

  We first want tp reorder both the reads e^1, e^2 to be outside the loop, naming it Candidate $C^{2\prime\prime}$. Using Condition 5.7, by Corollary 4.1.2, we can infer that $C^{2\prime\prime}$ has observable behaviors as a subset of C^2. From Condition 5.6, by Theorem 5.1, we can eliminate either e^1 or e^2, thus resulting in $C^{2\prime}$ whose observable behaviors is a subset of $C^{2\prime\prime}$. By transitive property of subsets we can infer that $C^{2\prime}$ 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 $C^{n\prime}$ 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 $C^{n+1\prime\prime}$.

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

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, not being a conditional check. Consider program P' with event e agent ordered after 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 $C^{1\prime}$ such that

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

has observable behaviors as a subset of C^1

For every $k \in K$ such that $e \xrightarrow[ao]{} k$ we need them to be reorderable with respect to e to bring it outside the loop. Using Condition 5.7, we have from Corollary 4.1.2

that $C^{1\prime}$ 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. Using Condition 5.8, we have from Prop 1 that e is not part of any conditional branch. Thus, from Corollary 4.1.4, we can infer that the transformed program has observable behaviors a subset of the original.

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 $C^{2\prime\prime}$ to $C^{2\prime}$, note that in $C^{2\prime\prime}$ 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 $C^{2\prime}$ whose observable behaviors is a subset of $C^{2\prime\prime}$.

  By transitive property of subsets we can infer that $C^{2\prime}$ 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 $C^{n+1\prime\prime}$. From our inductive assumption, we can then

conclude that $C^{n+1\prime}$ has observable behaviors as a subset of C^n. By transitive property of subsets, we can infer that $C^{n+1\prime}$ has observable behaviors as a subset of C^{n+1}.

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.

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 $C^{1\prime}$ 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. Using Condition 5.13, we have from Corollary 4.1.2 that $C^{1\prime}$ 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. Using Condition 5.14, we have from Prop 1 that e is not part of any conditional branch. Thus, from Corollary 4.1.4, we can infer that the transformed program has observable behaviors a subset of the original.

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 $C^{2\prime}$ with just one such event e, such that:

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

We first reorder both the writes e^1, e^2 to be outside the loop, naming it Candidate $C^{2\prime\prime}$. Because we have Condition 5.10, by Corollary 4.1.2, we can infer that $C^{2\prime\prime}$ has observable behaviors as a subset of C^2. To go from $C^{2\prime\prime}$ to $C^{2\prime}$, note that in $C^{2\prime\prime}$ 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 $C^{2\prime}$ whose observable behaviors is a subset of $C^{2\prime\prime}$[4].

By transitive property of subsets we can infer that $C^{2\prime}$ 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 $C^{n\prime}$ 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

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

  By Corollary 4.1.2, we can infer that $C^{n+1\prime\prime}$ 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 $C^{n+1\prime\prime}$. From our inductive assumption, we can then conclude that $C^{n+1\prime}$ has observable behaviors as a subset of C^n. By transitive property of subsets, we can infer that $C^{n+1\prime}$ has observable behaviors as a subset of C^{n+1}.

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

□

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*

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

[4]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.

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 $C^{1\prime}$ such that

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

has observable behaviors as a subset of C^1

For every $k \in K$ such that $e \xrightarrow[ao]{} k$ we need them to be reorderable with respect to e to bring it outside the loop. Using Condition 5.2.7, we have from Corollary 4.1.2 that $C^{1\prime}$ 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. Using Condition 5.2.7, we have from Prop 1 that e is not part of any conditional branch. Thus, from Corollary 4.1.4, we can infer that the transformed program has observable behaviors a subset of the original.

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 e^1 and e^2. We need candidate $C^{2\prime}$ 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 $C^{2\prime\prime}$ 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 $C^{2\prime}$ whose observable behaviors as a subset of $C^{2\prime\prime}$.

  By transitive property of subsets we can infer that $C^{2\prime}$ 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 $C^{n\prime}$ 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

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

Having Condition 5.2.7, by Corollary 5.2.1, we can eliminate e^n, thus giving us candidate C^n whose observable behaviors are a subset of C^{n+1}

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

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 for loop-invariant code motion are valid for whenever it 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 across nesting as well as from one loop to another. We cannot prove when we can reorder events across 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. 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 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 that are important.

Separation of Concerns Our approach to program transformations avoids the algorithmic complexity of program transformations. We do not assume why the

compiler would perform such a transformation. While this benefits our analysis to be independent 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* (the conditional variable may be local to the thread). 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 4.1.4, 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 for different optimizations. 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 and even design new ones incorporating the impact of the model.

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. 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, is 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 memory accesses. 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 remaining road-map we had in mind during the 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(RMW) events exist in programs. However, this assumption, in general, is too strong(eg: Compare-and-Swap, Atomic Increment/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 in terms of shared memory accesses.

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 independent 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 considered. 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 accesses. However, their semantics from a consistency model perspective is given to be same as that of synchronize events. 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 across loops. This will also give an interesting equivalence to instruction reordering. Other program transformations we find important to consider are strengthening/ weakening access modes (fence optimizations), gathering optimization(merging small size accesses), changing tearing factor of accesses.

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 under-specified 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 language 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 ECMAScript language level, we declare an access to tear despite the hardware having it to be tear-free, this would 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 in Figure 6.1, 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.

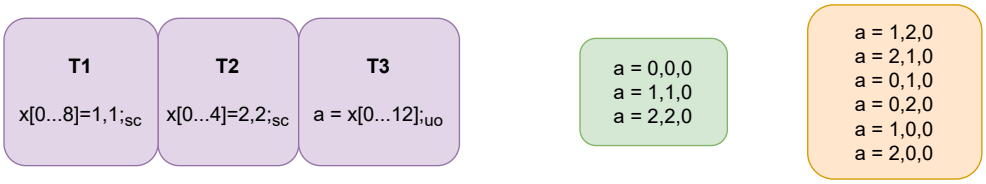

Figure 6.1: Mixed-size Tearing reads example

The green box represents the outcomes that we can intuitively justify. 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 existence 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-size is uncertain. This affects the way we can reason with our programs and their corresponding observable behaviors.

Consider the two candidates in Figure 6.2, each of which could represent the same program, one where the initialize event is to the whole 8-byte(right) and one where its split into two events of type *init* writing 4 bytes each(left). The orange box is an observable behaviors in question.

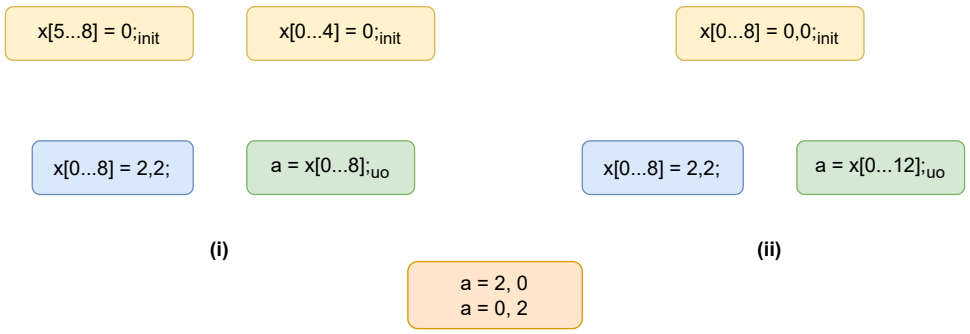

Figure 6.2: Mixed-size Init events example

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

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

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 in Figure 6.3 with the orange box having the observable behavior in question.

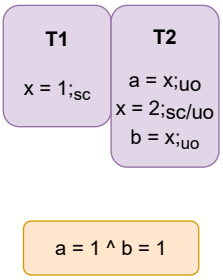

Figure 6.3: Coherence violation Example.

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 outcome 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 in Figure 6.4 with a few mixed size accesses, where the writes are of type *sc*.

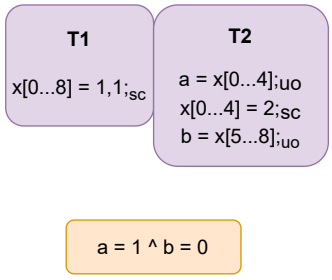

Figure 6.4: Mixed-size events example where coherence is assumed to be held.

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 now 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 [20] 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 ([20] 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 assessing 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 [39] try to do this for Total-Store-Order (TSO) and a fraction of Power memory model. They do this by describing a transformational model over SC.

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 [40] 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 the executions of our programs, forcing us to move away from sequential reasoning. With the introduction of relaxed memory consistency, this sequential reasoning became more of a hurdle. The need for non-sequential weak memory reasoning is still not something that is viewed by many necessary, 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 intuitive 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 sincerely hope that this thesis helps in paving way towards that direction.

Bibliography

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

[2] A. Gopalakrishnan and C. Verbrugge, "Reordering under the ecmascript memory consistency model," *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, vol. (In Press), 2020.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] J. Alglave, D. Kroening, J. Lugton, V. Nimal, and M. Tautschnig, "Soundness of data flow analyses for weak memory models," in *Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings*, H. Yang, Ed., ser. Lecture Notes in Computer Science, vol. 7078, Springer, 2011, pp. 272–288. DOI: 10.1007/978-3-642-25318-8_21. [Online]. Available: https://doi.org/10.1007/978-3-642-25318-8%5C_21.

[25] J. Ševčík 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.

[26] 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.

[27] 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.

[28] J. Ševčík, "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.

[29] 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.

[30] 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.

[31] A. Podkopaev, O. Lahav, and V. Vafeiadis, "Bridging the gap between programming languages and hardware weak memory models," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 69:1–69:31, 2019. DOI: 10.1145/3290382. [Online]. Available: https://doi.org/10.1145/3290382.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] D. Lustig, A. Wright, A. Papakonstantinou, and O. Giroux, "Automated synthesis of comprehensive memory model litmus test suites," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Y. Chen, O. Temam, and J. Carter, Eds., ACM, 2017, pp. 661–675. DOI: 10.1145/3037697.3037723. [Online]. Available: https://doi.org/10.1145/3037697.3037723.

Appendix A

Counter Examples

A.1 Examples of cases where reordering at the candidate level may not be safe.

For all the examples we show here, we only show the ordering relations that are important to observe. We exhaustively show for all the cases from Table 4.23 where reordering is not safe to do. Some of these cases are addressed together as the same counter example suffices to show they are unsafe to do.

Reads to same memory where e is of type sc and d is 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 A.1 shows an example of a candidate (left) along with with its candidate execution (right) where the case of outcome in the red box is not possible.

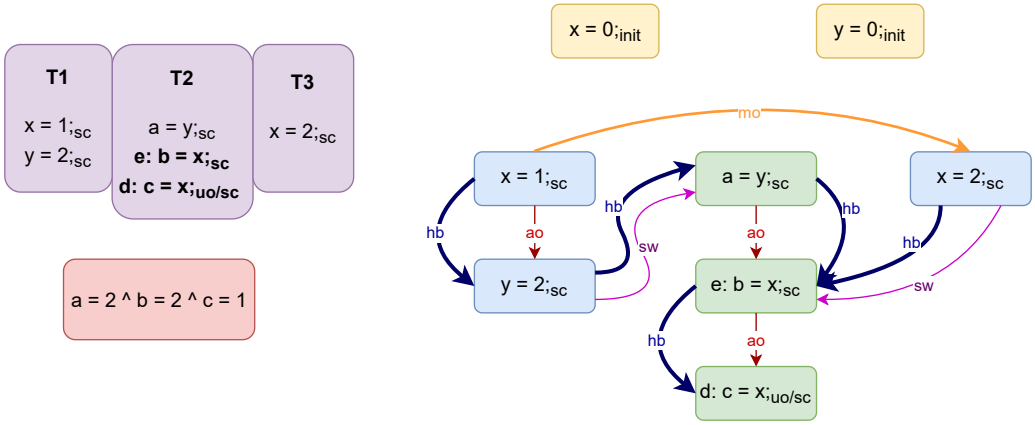

Figure A.1: Case where $a = 2, b = 2, c = 1$ is invalid due to Sequentially Consistent Atomics

- For $a = 2$, it must come from the write $y = 2;_{sc}$. Thus, we would have $\{y = 2;_{sc}\} \overrightarrow{sw} \{a = y;_{sc}\}$ and hence $\{y = 2;_{sc}\} \overrightarrow{hb} \{a = y;_{sc}\}$.
- For $b = 2$, it must come from the write $x = 2$. Thus, we would have $\{x = 2;_{sc}\} \overrightarrow{sw} \{b = x;_{sc}\}$ and hence $\{x = 2;_{sc}\} \overrightarrow{hb} \{b = x;_{sc}\}$.
- Using $\overrightarrow{ao}$ and the above result, we can infer that $\{x = 2;_{sc}\} \overrightarrow{mo} \{x = 2;_{sc}\}$, $\{x = 1;_{sc}\} \overrightarrow{hb} \{c = x;_{uo/sc}\}$ and $\{x = 2;_{sc}\} \overrightarrow{hb} \{c = x;_{uo/sc}\}$.
- These relations correspond to patterns of Axiom 3, from which we can conclude that we cannot have $\{c = x;_{uo/sc}\} \overrightarrow{rf} \{x = 1;_{sc}\}$.

Hence, the observable behavior $a = 2 \ \wedge \ b = 2 \ \wedge \ c = 1$ is disallowed.

Figure A.2 shows the Candidate (right) after reordering the two reads in $T2$ along with its candidate execution (left) where the same case of reads(orange box) is possible.

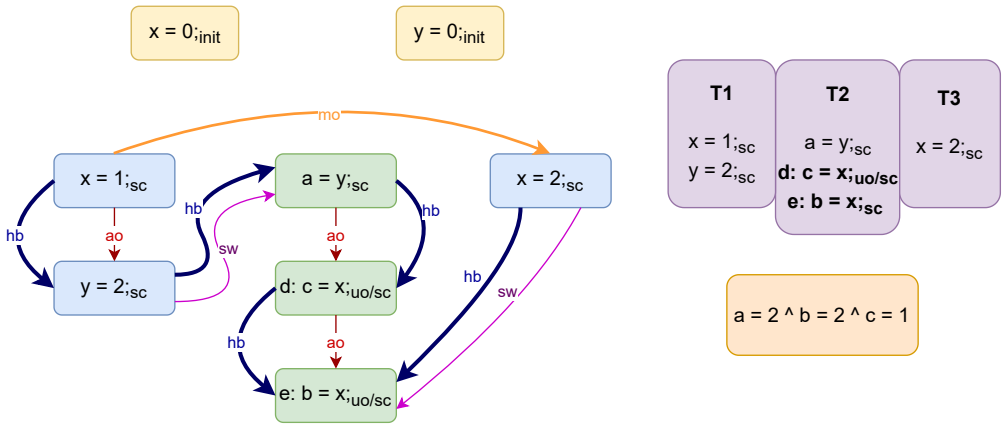

Figure A.2: Case where the reads are reordered and $a = 2, b = 2, c = 1$ is valid.

- For $a = 2$, it must come from the write $y = 2_{sc}$. Thus, we would have $\{y = 2;_{sc}\} \overrightarrow{sw} \{a = y;_{sc}\}$ and hence $\{y = 2;_{sc}\} \overrightarrow{hb} \{a = y;_{sc}\}$.
- From this, we also get $\{x = 1;_{sc}\} \overrightarrow{hb} \{c = x;_{uo/sc}\}$ and $\{x = 1;_{sc}\} \overrightarrow{hb} \{b = x;_{sc}\}$.
- Now, for $c = 1$, it must come from the write $\{x = 1;_{sc}\}$. This is possible as the above relations do not represent any pattern that restricts $\{c = x;_{uo/sc}\} \overrightarrow{rf} \{x = 1;_{sc}\}$. .
- Now for $b = 2$, it must come from the write $\{x = 2;_{sc}\}$. Since there is no happens-before or memory-order relations with this write, none of the axioms would restrict $\{b = x;_{sc}\} \overrightarrow{rf} \{x = 2;_{sc}\}$.

Thus, the observable behavior $a = 1 \ \wedge \ b = 2 \ \wedge \ c = 2$ is allowed by the program after reordering.

A Read e of type sc followed by a Write of either uo/sc Figure A.3 shows an example of a candidate (left) along with with its candidate execution (right) where the case of outcome in the red box is not possible.

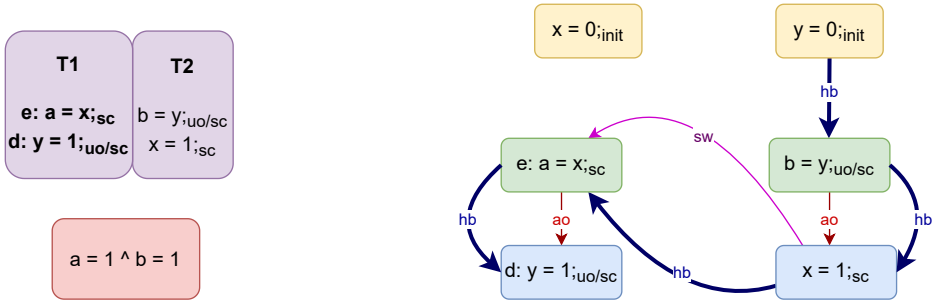

Figure A.3: Case where $a = 1, b = 1$ is invalid due to Coherent Reads.

Observations:

- For $a = 1$, it must come from the write $x = 1_{sc}$. Thus, we would have $\{x = 1;_{sc}\} \overrightarrow{sw} \{a = x;_{sc}\}$ and hence $\{x = 1;_{sc}\} \overrightarrow{hb} \{a = x;_{sc}\}$.
- From the above relation and using $\overrightarrow{ao}$, we can infer $\{b = y_{uo/sc}\} \overrightarrow{hb} \{y = 1_{uo/sc}\}$
- The above relation corresponds to a pattern of Axiom 1, from which we can infer that we cannot have the relation $\{b = y;_{uo/sc}\} \overrightarrow{rf} \{y = 1;_{uo/sc}\}$.

Hence, the observable behavior $a = 1 \ \wedge \ b = 1$ is disallowed.

Figure A.4 shows the Candidate after reordering(right) a read and write in $T1$ along with its candidate execution(left) where the same case of reads(orange box) is possible.

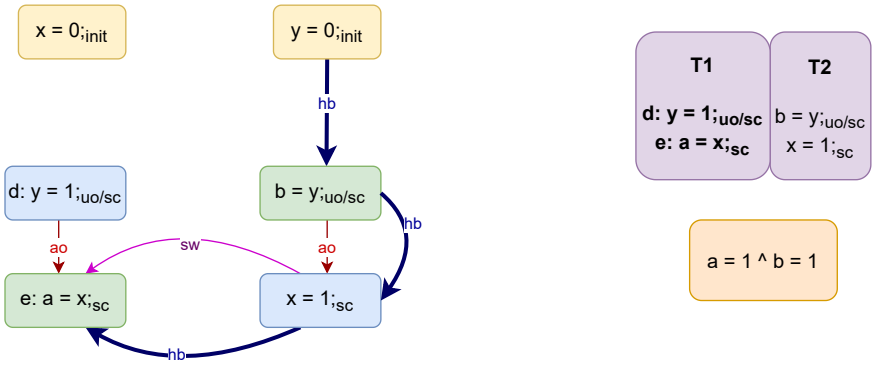

Figure A.4: Case where events of T1 are reordered, resulting in $a = 1, b = 1$ to be valid.

Observations:

- For $a = 1$, it must come from the write $x = 1_{sc}$. Thus, we would have $\{x = 1;_{sc}\} \xrightarrow[sw]{} \{a = x;_{sc}\}$ and hence $\{x = 1;_{sc}\} \xrightarrow[hb]{} \{a = x;_{sc}\}$.
- From the above, we cannot infer $\{b = y_{uo/sc}\} \xrightarrow[hb]{} \{y = 1_{uo/sc}\}$. In fact no happens-before relation can exist between the two events.
- We can have the relation $\{b = y;_{uo/sc}\} \xrightarrow[rf]{} \{y = 1;_{uo/sc}\}$.

Hence, the observable behavior $a = 1 \ \wedge \ b = 1$ is allowed.

A Read e of type uo followed by a write d of type sc For this we can use the same example in Figure A.3 where we just reorder $T2$'s events. We leave justifying the new observable behavior introduced as an exercise for the reader.

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 after reordering creates a $\xrightarrow[hb]{}$ cycle. Figure A.5 is one such example with its Candidate and Corresponding Candidate Execution in question:

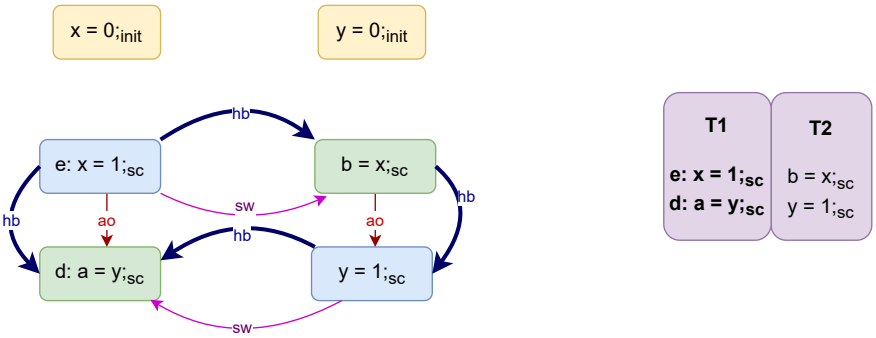

Figure A.5: Case where $a = 1, b = 1$ is valid and no happens-before cycles

Figure A.6 shows the Candidate after reordering(right) the two events in $T1$, which makes the same candidate execution(right) as above contain a happens-before cycle:

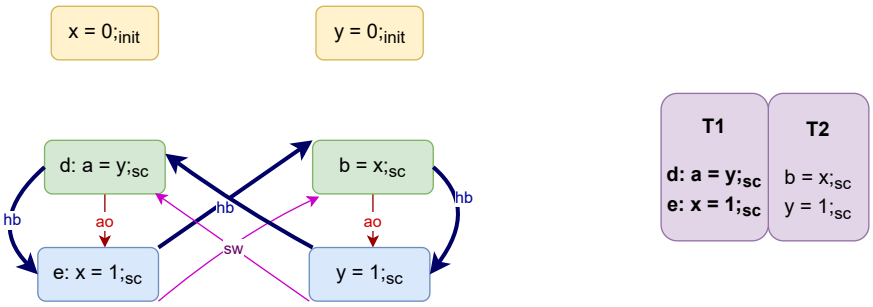

Figure A.6: Case where $a = 1, b = 1$ implies a happens-before cycle

Observation:

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

One might think that simply discarding the above Candidate execution would do. But this would mean discarding certain $\xrightarrow{}_{hb}$ relations by considering them to be irrelevant; this would require more information to infer which relations are going to create such cycles and which are not. In practice a cycle may span several events from different threads. Since we place no assumptions on these relations. Hence, the following reordered program outcome is something we do not risk to allow.

A Write e of type uo/sc followed by a Write d of type sc Figure A.7 shows an example of a candidate(left) along with with its candidate execution(right) where the case of outcome in the red box is not possible.

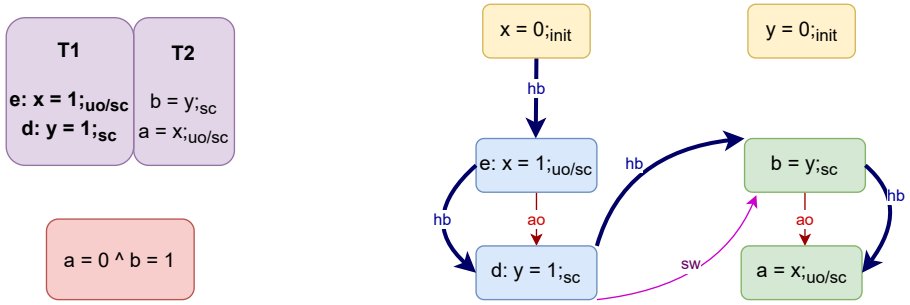

Figure A.7: Case where $a = 0, b = 1$ is invalid due to Coherent Reads.

Observations:

- For $b = 1$, it must come from the write $y = 1_{sc}$. Thus, we would have $\{y = 1;_{sc}\} \overrightarrow{sw} \{b = y;_{sc}\}$ and hence $\{y = 1;_{sc}\} \overrightarrow{hb} \{b = y;_{sc}\}$.
- From the above relation, we can infer $\{x = 0_{init}\} \overrightarrow{hb} \{x = 1_{uo/sc}\} \overrightarrow{hb} \{a = x_{uo/sc}\}$.
- The above relation corresponds to a pattern of Axiom 1, from which we can conclude that we cannot have $\{a = x;_{uo/sc}\} \overrightarrow{rf} \{x = 0;_{init}\}$.

Hence, the observable behavior $a = 0 \ \wedge \ b = 1$ is disallowed.

Figure A.8 shows the Candidate after reordering(right) the two writes in $T1$ along with its candidate execution(left) where the same case of reads(orange box) is possible.

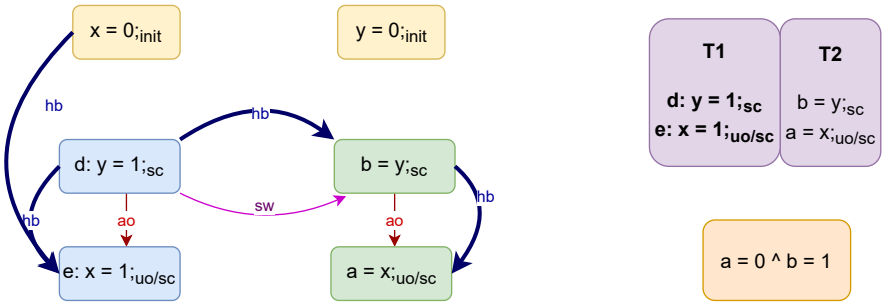

Figure A.8: Case where events of T1 are reordered, resulting in $a = 0, b = 1$ to be valid.

Observations:

- For $b = 1$, it must come from the write $y = 1_{sc}$. Thus, we would have $\{y = 1;_{sc}\} \overrightarrow{sw} \{b = y;_{sc}\}$ and hence $\{y = 1;_{sc}\} \overrightarrow{hb} \{b = y;_{sc}\}$.
- From the above relation and $\overrightarrow{ao}$, we cannot infer $\{x = 1_{uo/sc}\} \overrightarrow{hb} \{a = x_{uo/sc}\}$.
- Thus, we can have $\{a = x;_{uo/sc}\} \overrightarrow{rf} \{x = 0;_{init}\}$.

Hence, the observable behavior $a = 0 \ \wedge \ b = 1$ is allowed.

A.2 Examples where reordering across conditional branches may not be safe.

Case where we have a 2-branch conditional Figure A.9 shows an example of a program with conditionals (left) along with with its candidate execution(right) where the left conditional branch is taken and the outcome in the red box is not possible.

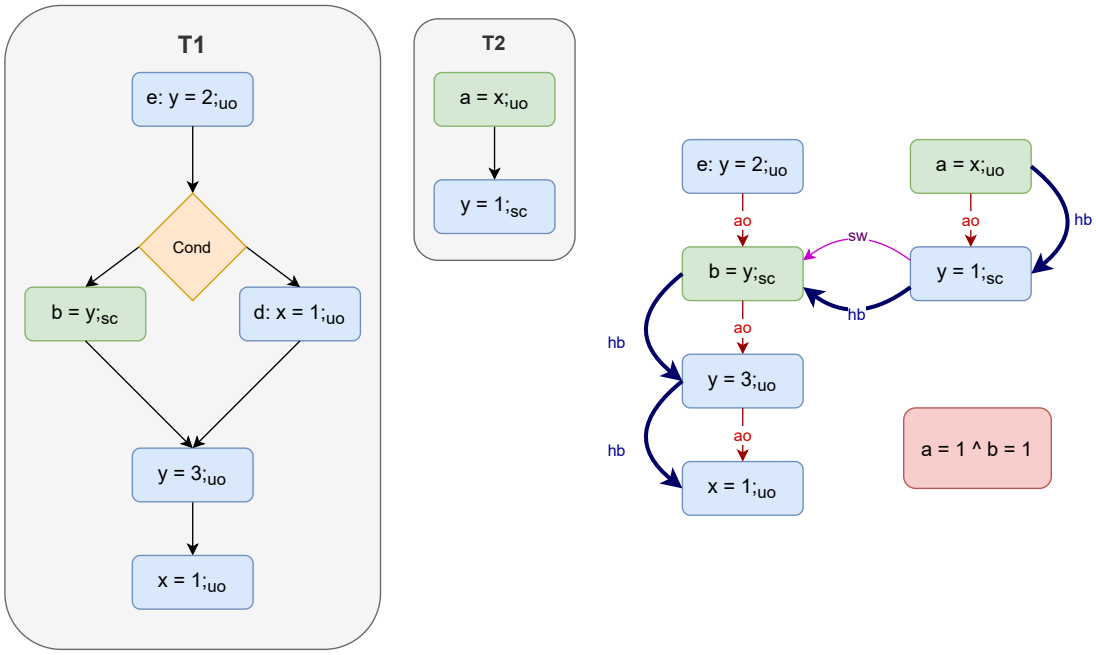

Figure A.9: Case with conditionals where $a = 1, b = 1$ is invalid due to Coherent Reads.

Observations:

- For $b = 1$, it must come from the write $y = 1_{sc}$. Thus, we would have $\{y = 1;_{sc}\} \overrightarrow{sw} \{b = y;_{sc}\}$ and hence $\{y = 1;_{sc}\} \overrightarrow{hb} \{b = y;_{sc}\}$.
- Using this relation and $\overrightarrow{ao}$, we can infer $\{a = x;_{uo}\} \overrightarrow{hb} \{x = 1;_{uo}\}$.
- The above relation corresponds to Axiom 1, from which we conclude that we cannot have $\{a = x;_{uo}\} \overrightarrow{rf} \{x = 1;_{uo}\}$.

Figure A.10 shows the program after reordering(right) two writes in $T1$ along with its candidate execution(left) where the same left conditional branch is taken and the outcome in the orange box is possible.

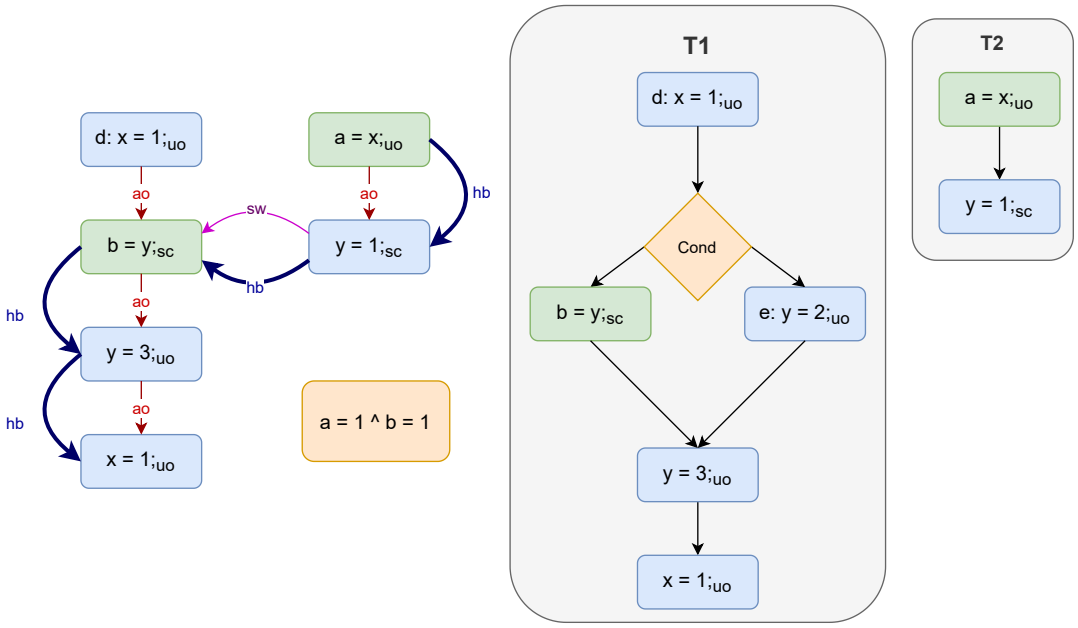

Figure A.10: Case where events of T1 are reordered, resulting in $a = 1, b = 1$ to be valid.

Observations:

- For $b = 1$, it must come from the write $y = 1_{sc}$. Thus, we would have $\{y = 1;_{sc}\} \overrightarrow{sw} \{b = y;_{sc}\}$ and hence $\{y = 1;_{sc}\} \overrightarrow{hb} \{b = y;_{sc}\}$.
- Now there is also event d, but there is no $\overrightarrow{hb}$ relation between d and $\{a = x_{uo}\}$.
- Thus for $a = 1$, we can obtain this from event d. Thus, we can have the relation $\{a = x;_{uo}\} \overrightarrow{rf} \{d : x = 1;_{uo}\}$.

Case with 1-branch conditional Figure A.11 shows an example of a program (left) with 1-branch conditional along with with its candidate execution (right) where the conditional branch is not taken and the outcome in the red box is not possible.

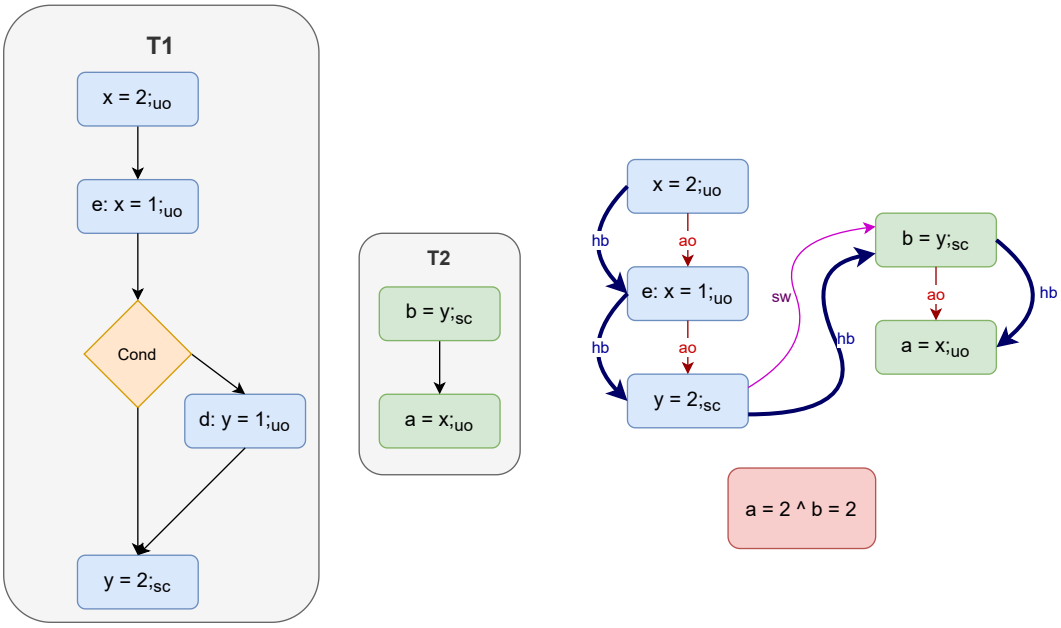

Figure A.11: Case with conditionals where $a = 2, b = 2$ is invalid due to Coherent Reads.

Observations:

- For $b = 2$, it must come from the write $y = 2_{sc}$. Thus, we would have $\{y = 2;_{sc}\} \overrightarrow{sw} \{b = y;_{sc}\}$ and hence $\{y = 2;_{sc}\} \overrightarrow{hb} \{b = y;_{sc}\}$.
- From the above relation and $\overrightarrow{ao}$, we can infer $\{x = 2;_{uo}\} \overrightarrow{hb} \{e : x = 1;_{uo}\} \overrightarrow{hb} \{y = 2;_{sc}\} \overrightarrow{hb} \{b = y;_{sc}\} \overrightarrow{hb} \{a = x;_{uo}\}$.
- The above relation is a pattern of Axiom 1, using which we can conclude that we cannot have $\{a = x;_{uo}\} \overrightarrow{rf} \{x = 2;_{uo}\}$.

Figure A.12 shows the program after reordering (right) two writes in $T1$ along with its candidate execution (left) where the conditional branch is not taken and the outcome in the orange box is possible[1].

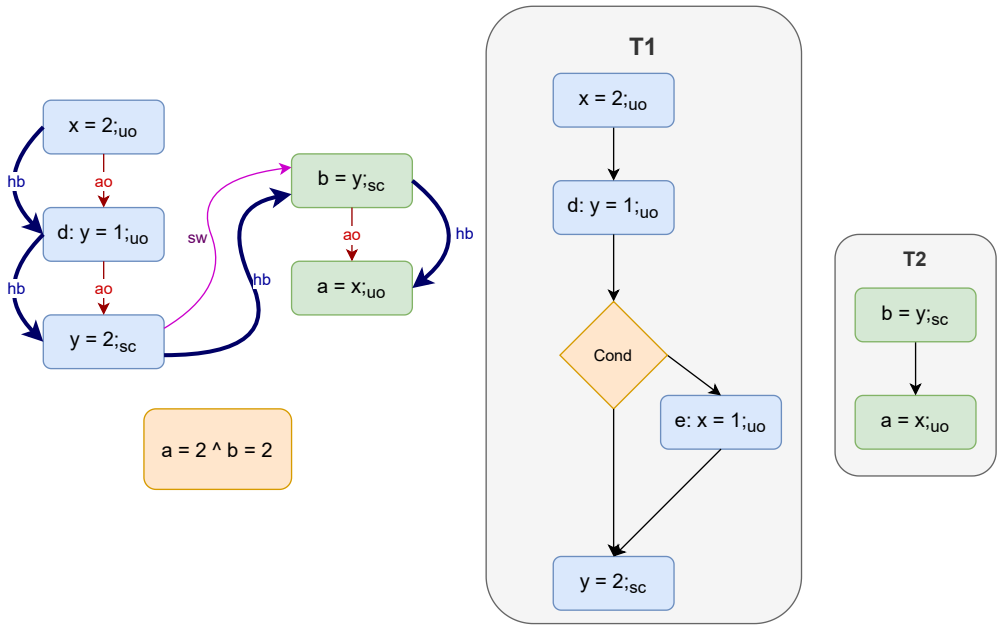

Figure A.12: Case where events of $T1$ are reordered, resulting in $a = 2, b = 2$ to be valid.

Observations:

- For $b = 2$, it must come from the write $y = 2_{sc}$. Thus, we would have $\{y = 2;_{sc}\} \overrightarrow{sw} \{b = y;_{sc}\}$ and hence $\{y = 2;_{sc}\} \overrightarrow{hb} \{b = y;_{sc}\}$.
- Because event e is now within the conditional branch, we cannot have $\{e : x = 1;_{uo}\} \overrightarrow{hb} \{a = x;_{uo}\}$.
- In addition, we have using the above inferred relation $\{x = 2;_{uo}\} \overrightarrow{hb} \{a = x;_{uo}\}$.
- Hence, we can have the relation $\{a = x;_{uo}\} \overrightarrow{rf} \{x = 2;_{uo}\}$.

We leave the rest of the cases as an exercise to the avid reader[2].

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

[2]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 reading a shared memory which must have not been there because a conditional branch was not taken. This fact, however, does not rely on the consistency rules of the memory model. Possible reason such a reordering helps could be 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 will return the same constant value. Having such an assertion in general might not be always certain. This is one of the drawbacks of not using information as to why the compiler would do such a transformation.