Reordering Under the ECMAScript Memory Consistency Model

Akshay Gopalakrishnan and Clark Verbrugge

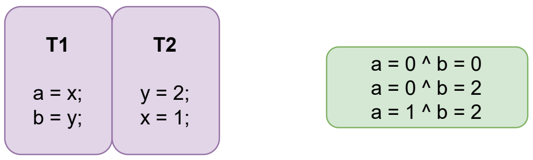
McGill University
Montréal, Québec, Canada
`akshay.akshay@mail.mcgill.ca`, `clump@cs.mcgill.ca`

Abstract. Relaxed memory accesses are used to gain substantial improvement in the performance of concurrent programs. A relaxed 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 compiler optimizations essential for the performance of programs overall. In this paper, we give a formal description of the ECMAScript relaxed memory consistency model. We then analyze the impact of this model on one of the most common compiler optimizations, viz. *instruction reordering*. We give a conservative proof under which such optimization is allowed for relaxed memory accesses. Finally, we discuss the advantage of our conservative approach and the gaps needed to be filled in order to incorporate such analysis while doing such optimizations at the program level.

Keywords: relaxed memory consistency, optimization, ECMAScript

1 Introduction

Concurrent programs take advantage of *out-of-order* execution. Intuitively, this means that more than one unrelated computations can be done "simultaneously" without having any fixed order in which they should happen. This results in concurrent programs having multiple different outcomes, the possible outcomes of which are described by a *memory consistency model*. The most intuitive and commonly relied upon model is that of *Sequential Consistency* (SC), which guarantees that every outcome of a program must be equivalent to a sequential interleaving of each thread's individual actions. For example, consider the program in Figure 1 with two threads, which share memory denoted by x, y initialized to 0, where a, b are local variables. The right-hand-side are the possible values that a and b can read under sequential consistency rules.



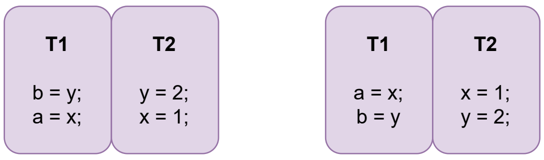
```
a = x;
b = y;
```

```
y = 2;
x = 1;
```

a = 0 ^ b = 0
a = 0 ^ b = 2
a = 1 ^ b = 2

Fig. 1. Example program with its possible outcomes under sequential consistency.

However, the above program under SC cannot have the outcome $a{=}2 \wedge y{=}0$. From a program transformation standpoint, such an outcome should be possible: we can simply reorder either both the reads or both writes to x and y, as they are computations on disjoint memory. But from a consistency rule standpoint, since the outcome is not valid, it also brings with it the conclusion that such simple program transformations may not be safe or even invalid. Figure 2 shows how after doing either one of these reorderings, an outcome invalid under SC is possible.



T1	T2

b = y;
a = x;
	y = 2;
	x = 1;

T1	T2

a = x;
b = y
	x = 1;
	y = 2;

Fig. 2. Left program is when the two reads in T1 are reordered, whereas the right program is when the two writes of T2 are reordered.

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

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

1. We provide a concise *declarative style* model of the core ECMAScript memory consistency semantics. This clarifies the existing draft presentation [4] in a manner useful for validating optimizations.
2. Using our model we show when basic reordering of independent instructions is allowed. Although conservative, this represents a formal proof that this fundamental optimization is permitted. Similar proof designs can be used to validate other basic optimization behaviours,such as removing redundant reads or writes.

2 Related Work

Sequential Consistency, which was first formulated by Lamport et al. [5], gives programmers a very intuitive way to reason about their programs running in a multiprocessor environment. However, in the practical sense, Sequential Consistency is too "strict," in the sense that it may impede possible performance benefits of using low level optimization features, such as instruction reordering, or read/write buffers provided by the hardware. A tutorial by Adve et al. [1], summarizes the most common hardware features for relaxed memory that are now available in most hardware. What this tutorial also exposed is the difficulty in formalizing such features in a way that we can reason about our programs sanely without getting caught up in the complexity of multiple executions of our programs. Unsurprisingly, relaxed memory model specifications for different hardware / high level programming languages are still sometimes written in informal prose format, which lead to a number of problems in implementation [12].

Sarkar et al. [9] showed that the original x86-CC memory model was fairly informal, which they then formalized in their work. This also exposed inconsistencies between the specification and the implementation in hardware. This was shown in their subsequent work done by Owens et al. [8], wherein they proposed a new memory model x86-TSO as a remedy. Manson et al. [6], showed that the initial specifications of the Java memory model were quite informal and ill defined, and offered a more precise formalization. Recent works such as that done by Bender et al. [3], also shows us that the recent updates to the java Memory model is still relatively unclear, which they again formalize. Similarly, Batty et al. [2], clarified the specification of the C11 memory model.

Apart from the problems of ill defined / informal specifications, these models also have an impact on the safety of program transformations which were considered safe to do in a sequential program. Ševčík et al. [11] showed that standard compiler optimizations were rendered invalid under the respective memory model of Java. Vafeiadis et al. [13] showed that common compiler optimizations under C11 memory model are also invalid, followed by proposing some changes to allow them.

With respect to instruction reordering in shared memory programs, Ševčík et al. [10] recently gave a proof design on how to show such optimizations are valid. However, this approach relies on the idea of reconstructing the original execution of a program given the optimized one, while also showing the well known SC-DRF guarantee holds—programs that are *data race free* (DRF) must exhibit SC semantics. Our approach is in fact the other way round; we show that the optimized program does not introduce new behaviours, by explicitly using the consistency rules to show that relevant ordering relations are preserved.

ECMAScript has also had some attention in this respect. Watt et al [14] uncovered and fixed a deficiency in the previous version of the model, repairing the model to guarantee SC-DRF. 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.

3 The ECMAScript Memory Consistency Model

We give a relatively more formal and concise axiomatic description of Section 28 of the ECMAScript standard. The version we are referring to is the current working draft [4]. It is important to note that this working draft has not changed the memory model specifics since the time we started our work on this.

3.1 Agents and Agent Clusters

Agents for our context could be thought analogous to different threads/processes running concurrently. Every agent is mapped to a list of events. (defined below) Collection of agents using a common shared memory for communication form an agent cluster. There can be multiple agent clusters, however, an agent can only belong to one agent cluster.

3.2 Events

Agent execution is modelled in terms of events. An event, in our context, is either an operation that involves (shared) Read/Write memory access or Synchronize events that constrains the order of execution of multiple events. We define $\boldsymbol{E}$ as the set of events involved in an agent cluster. We refer to $\boldsymbol{SM}$, $\boldsymbol{R}$, $\boldsymbol{W}$, $\boldsymbol{S}$ as sets of Shared Memory, Read, Write and Synchronize events respectively. Shared Memory events are composed of Read and Write event sets. Read-Modify-Write events belong to both $\boldsymbol{R}$ and $\boldsymbol{W}$.

Range ($\Re$) Each of the *shared memory events* are associated with a contiguous range of memory on which it operates. $\Re$ is a function that maps a shared memory event to the range of memory indices it operates on which we represent as a starting index i and a size s. As an example, the range of event e would be like:

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

We define two binary operators $\cap_\Re$ and $\cup_\Re$ to give the intersection and union respectively of the set of the byte indices, in order to describe disjoint, overlapping and equal ranges.

Types of events based on Order There are 3 types (or access modes) which play a role in the sequence in which event actions are visible to different agents

1. **Sequentially Consistent** (sc) - Events of this type are *atomic* in nature. There is a strict global total ordering of such events which is agreed upon by all agents in the agent cluster.
2. **Unordered** (uo) - Events of this type are considered non-atomic and can occur in different orders for each agent.
3. **Initialize** ($init$) - Events of this type are used to initialize the values in memory ordered before events in an agent cluster.

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 a figure, the type would be represented as a subscript: $event_{type}$.

Tearing (or not) Additionally, each shared-memory event is also associated with a tearing factor. 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 access. The implication of tearing is better understood with the consistency rule that will later be shown.

3.3 Relation among events

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

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

Read-Bytes-From ($\xrightarrow{rbf}$) This relation maps every read event to a list of tuples consisting of a write event and the corresponding byte index that is read. For instance, consider a read event e and corresponding write events $d1$, $d2$ all of whose ranges have byte index i and size 3. One possible $\xrightarrow{rbf}$ relation could be represented as

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

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

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

Reads-From ($\xrightarrow{rf}$) This relation, is similar to the above relation, except that the byte index details are not involved in the composed list. So for the above example, the rf relation would be represented either as $e \xrightarrow{rf} (d1,d2)$ or individual binary read-write relation as $e \xrightarrow{rf} d1$ and $e \xrightarrow{rf} d2$.

Agent-Synchronizes With (*ASW*) A list for each agent that consist of ordered tuples of synchronize events. These tuples specify ordering constraints among agents at different points of execution. We represent such a list for an agent k as

$$\boldsymbol{ASW}_k = \{\langle s_1,s_2\rangle,\langle s_3,s_4\rangle ...\}$$

3.4 Ordering Relations among Events

Agent Order ($\xrightarrow{ao}$) A total order 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 $e \xrightarrow{ao} d$ or $d \xrightarrow{ao} e$.

Synchronize-With Order ($\xrightarrow{sw}$) Represents the synchronizations among different agents through relations between their events. It is a composition of two sets as below:

$$\forall i,j > 0,\ \langle s_i,s_j\rangle \in ASW \ \Rightarrow\ s_i \xrightarrow{sw} s_j$$
$$(e \xrightarrow{rf} d) \wedge e\!:\!sc \wedge d\!:\!sc \wedge (\Re(e) = \Re(d)) \ \Rightarrow\ (d \xrightarrow{sw} e)$$

Happens Before Order ($\xrightarrow{hb}$) A transitive order on events, composed of the following:

$$e \xrightarrow{ao} d \ \Rightarrow\ e \xrightarrow{hb} d$$
$$e \xrightarrow{sw} d \ \Rightarrow\ e \xrightarrow{hb} d$$
$$\forall e,d \in SM,\ e\!:\!init \wedge (\Re(e) \cap_{\Re} \Re(d) \neq \phi) \ \Rightarrow\ e \xrightarrow{hb} d$$

Memory Order ($\xrightarrow[mo]{}$) This is a total order on all events which respects happens-before

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

3.5 Some Preliminary Definitions

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

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

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

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

Definition 4. *Observable Behavior. The set of pairwise $\xrightarrow[rf]{}$ and $\xrightarrow[rbf]{}$ relations that result in one execution of the program. Think of this as our outcome of a program execution.*

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.

Coherent Reads There are certain restrictions of what a read event cannot see in an execution based on $\xrightarrow[hb]{}$ relation with write events.

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

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

A read (e) value cannot come from a write (d) that has happened after it or if there is a write (g) that happens between them, writing to the same memory:

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

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

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

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:

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

There are three separate cases that restrict e to read from d, which are as below:

- If all events are sequentially consistent with equal ranges.
- If both g and d are sequentially consistent with equal ranges and they happen before e.
- If both e and g are sequentially consistent with equal ranges and d happens before them.

The above cases can be summarized concisely by the rules below:

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

3.7 Race

Race Condition (*RC*) We define ***RC*** as the set of all pairs of events that are in a race. Two events e and d are in a race condition when they are shared memory events ($e,d \in \boldsymbol{SM}$), having overlapping ranges, not having a $\xrightarrow[hb]{}$ relation with each other, and which are either two writes or the two events are involved in a $\xrightarrow[rf]{}$ relation with each other. This can be stated concisely as,

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

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:

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

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

3.8 Consistent Executions (Valid Observables)

A valid observable behaviour is when:

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

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

4 Instruction Reordering

Instruction reordering is a common operation in compiler optimization, essential to instruction scheduling of course, but also implicit in loop invariant removal, partial redundancy elimination, and other optimizations that may move instructions. However, we saw previously how concurrent programs, under sequential consistency, may not be allowed to reorder certain events. Understanding precisely when we can safely reorder requires information on what instructions threads may be executing concurrently, which requires impractically expensive whole program analysis.

4.1 Our approach

Our solution to this is to construct a proof that would expose/specify the conditions under which reordering is possible given the relaxed memory semantics, while using information restricted to only the thread whose events are reordered. We construct the proof on candidate executions of a program. To keep things simple, assume that:

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

We first consider when consecutive events in the same agent can be reordered, followed by non-consecutive cases. The crux of the proof is to guarantee that reordering does not bring any new observable behaviors.

4.2 Preliminaries

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

Definition 5. *Consecutive pair of events (*cons*)*

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

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

Definition 6. *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 defined formally as*

$$\nexists \ g. \ e \xrightarrow[hb]{} g \wedge g \xrightarrow[hb]{} d$$

We can infer certain relations/conditions that must hold using this function based on some information on events e and d.

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

4.3 Lemmas to assist our proof

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

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

If

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

then,

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

Proof. We have the following to be true :

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

In both cases where d is unordered or a sequentially consistent write, for any event k

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

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

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

Note that although there could be a direct *happens-before* relation with some event k from *another* agent, they are only relations satisfying $dir(d,k)$.

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,

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

Proof. The proof is symmetric to that of Lemma 1.

Note that the above lemmas are only for events k which are not of type init

4.4 Valid Reordering

We view reordering as manipulating the agent-order relation among two events. In that sense, reordering two events e and d with $e \xrightarrow[ao]{} d$ effectively flips the relation around to $d \xrightarrow[ao]{} e$. What implications this change has on the other ordering relations depends what events e and d are and would require an analysis on each Candidate Execution. We begin by first defining a reorderable pair of events. We then formulate a theorem (with a proof) on the set of observable behaviors of a Candidate before and after reordering a pair of consecutive events which are reorderable. We consider reordering valid if the set of observable behaviours after reordering are a subset of the original.

Definition 7. *Reorderable Pair (Reord) We define a boolean function* Reord *that takes two ordered pair of events e and d such that e* $\xrightarrow[ao]{}$ *d and gives a boolean value indicating if they are a reorderable pair:*

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

Theorem 1. *Consider a candidate C of a program and its possible Candidate Executions where $\xrightarrow[hb]{}$ is strictly partial order. Consider two events e and d such that cons(e,d) is true in C and e $\xrightarrow[ao]{}$ d. Consider another candidate C' resulting after reordering e and d. Then if* Reord(e,d) *is true in C, the set observable behaviors possible due to Candidate Executions of C' is a subset of that of C.*

Proof. We look at this in terms of performing an instruction reordering on a candidate execution of C. We would want the resulting candidate execution to preserve all the other $\xrightarrow[hb]{}$ relations (except $e \xrightarrow[hb]{} d$) and that any new $\xrightarrow[hb]{}$ relations strictly reduce possible observable behaviors. This can be summarized as addressing four main questions for any *CandidateExecution* of C':

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

1. Preserving *happens-before* relations If some $\xrightarrow[hb]{}$ relations among events are missing in Candidate Executions of C' as compared to that of C, we may introduce new observable behaviors.

The relations that could be lost can be addressed by considering two disjoint sets of events in any *Candidate Execution* of C defined as below.

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

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

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

We consider $<p1,p2>$ as a pivot pair. This pair is *valid* if

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

The intuition is to *pivot* the $\xrightarrow[hb]{}$ relations to $p1$ and $p2$, such that after reordering e and d, we can "flow" the relations back to retain all of them (due to transitivity of happens-before).

By lemma 1, we have for C, the following condition on e where $p1$ is a valid pivot

$$e{:}\,uo \vee (e{:}\,sc \wedge e \in W).$$

Similarly, by lemma 2, we have for C, the following condition on d where $p2$ is a valid pivot

$$d{:}\,uo \vee (d{:}\,sc \wedge d \in R).$$

Table 1 summarizes the cases where we have a valid pair of pivots $<p1,p2>$

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

Table 1. Table summarizing whether we have valid pair of pivots based on e and d.

Note that the relations preserved are those other than $e \xrightarrow[hb]{} d$. Note also that relevant happens-before relations with initialize events are always preserved.

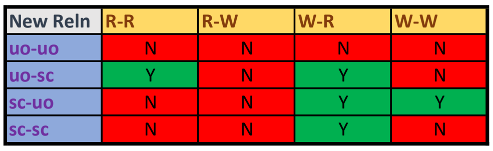
2. Additional *happens-before* relations Among cases that have valid pair of pivots, some may introduce new $\xrightarrow[hb]{}$ relations in Candidate Executions of C'. As an example, for the case when d is a sequentially consistent read, by lemma 1, in any Candidate Execution of C,

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

But in those of candidate C', by transitivity, we have

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

This is because there are sets of relations that come through $\xrightarrow[sw]{}$ relations. Table 2 summarizes the cases where new relations could be introduced, assuming valid pivot pairs.



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

Table 2. Table summarizing when new *happens-before* relations could be introduced based on having valid pair of pivots.

3. Presence of cycles? Before we go into analyzing whether new relations introduce observable behaviours, we first ensure there are no $\overrightarrow{hb}$ cycles introduced in the process. Note that if a cycle exists in Candidate Executions of C', then

1. The relations preserved do not themselves create a cycle
2. Additional new relations may introduce cycles

The first part is straightforward as we assume Candidate Executions of C have $\overrightarrow{hb}$ as a strict partial order. For the second part, we first address the case where $d \overrightarrow{hb} e$ may be part of the cycle. The other event k, may be either from the set K_e, K_d or a new relation that is formed.

1. Event k cannot belong to K_e or K_d, as by transitive property of $\overrightarrow{hb}$, a cycle would not exist.
2. For cases where $k \overrightarrow{hb} e$ is in the set of new relations, note that, $k \overrightarrow{hb} d$ already existed in the original Candidate Execution. On similar lines, for cases where $d \overrightarrow{hb} k$ is the set of new relations, $e \overrightarrow{hb} k$ exists. Thus, for both these cases also, a cycle with $d \overrightarrow{hb} e$ cannot exist.
3. For the last case where we have two new sets of relations formed, i.e $d \overrightarrow{hb} k$ and $k \overrightarrow{hb} e$, we could have a case where k is a common event for both sets. By lemma 1, we also have $k \overrightarrow{hb} d$ and by lemma 2, $e \overrightarrow{hb} k$. Thus, we have a cycle.

For the case when $d \overrightarrow{hb} e$ may not be part of the cycle, consider the first scenario where the new set of relations are of the form $k \overrightarrow{hb} e$. Suppose a cycle exists with another event k', then

$$k \overrightarrow{hb} e \wedge e \overrightarrow{hb} k' \wedge k' \overrightarrow{hb} k.$$

By lemma 1, we also have $k \overrightarrow{hb} d$ and by transitivity we also have $d \overrightarrow{hb} k'$. So, the following is also a cycle

$$k \overrightarrow{hb} d \wedge d \overrightarrow{hb} k' \wedge k' \overrightarrow{hb} k.$$

But these relations already existed in the original Candidate Execution, which implies a cycle existed in Candidate Execution of C. Thus, by contradiction, a cycle cannot exist. In similar lines we can show for the set $d \overrightarrow{hb} k$ that there cannot be any cycles.

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

On observing the role on the axioms on this relation, notice that if both e and d are read events then the range does not matter. For all other cases, if events e and d have overlapping ranges, one could introduce a new observable behavior after reordering them (a simple use of Coherent Reads / Sequentially Consistent Atomics). Any other new relations that are introduced can be divided into 4 cases, in terms of our events e and d and the new relation with some event k:

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

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

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

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

In each of the above cases, note that we need to only consider cases where their ranges are overlapping/equal.

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

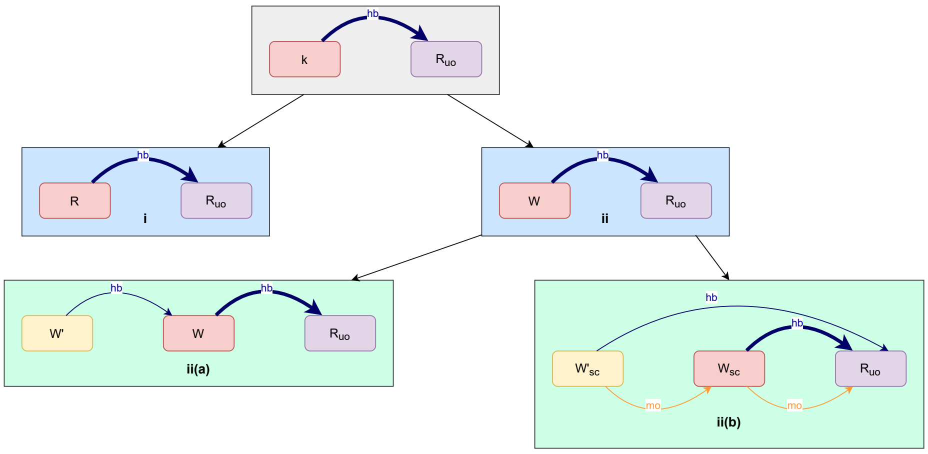


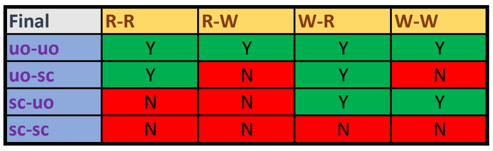
Fig. 3. The role of the axioms on introducing a new relation between an unordered read (purple box) and a preceding (by hb) event k (red box), varying based on whether k is a read, write, or sequentially consistent write.

1. For (i), when k is a read, none of the rules have any implications on observable behaviors.
2. For (ii), when k is a write, the rule of coherent reads (ii(a)) or sequentially consistent atomics (ii(b)) could restrict the read (e) from reading overlapping ranges of W' with W.

The above case analysis shows us that the new relation could 'trigger' the consistency rules, only to restrict possible reads-from relations, thus restricting possible observable behaviors.

Similarly, for other cases, the new relations could also 'trigger' the consistency rules, but again, only *restricting* $\xrightarrow[rf]{}$ relations.

Table 3 summarizes the valid cases where, we have a pair of valid pivots, where new relations do not introduce any $\xrightarrow[hb]{}$ cycles and may only restrict possible observable behaviors.



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

Table 3. 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 (Def. 7). □

Note that in the above we did not consider the $\xrightarrow[mo]{}$ relation, as preserving $\xrightarrow[hb]{}$ relations naturally preserves all the $\xrightarrow[mo]{}$ relations that must hold.

The following corollary helps us define when instruction reordering among non-consecutive events is possible.

Corollary 1. *Consider a Candidate C of a program and its Candidate Executions which are valid. Consider two events e and d where* $e \xrightarrow[ao]{} d$*, but* $\neg cons(e,d)$*. Consider another Candidate C' resulting after reordering e and d in C. Then, the set of Observable behaviors possible in C' is a subset of C only if Reord(e,d) and the following holds true.*

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

Proof. The proof is a straightforward induction on number of events k. □

5 Discussion

Theorem 1 and its corollary together give us a set of conditions that just need to be checked in addition while performing reordering of relaxed memory events. Having these set of conditions helps us avoid addressing the data-flow complexity due to different executions of the program using such accesses.

It is important to note that our approach is conservative, and one might be able to do reordering without causing new observable behaviors to occur even in cases that do not not satisfy our conditions. This is possible because certain happens-before relations may not be essential and hence discarding them will not result in any invalid observable behavior. Getting such information would require an analysis that takes into account relations that we cannot obtain using just intra-thread information, which in practice might be infeasible as the number of threads and events increase. (One such well studied analysis is May-Happen-In-Parallel, whose origins come from the work done by Naumovich et al. [7]).

It is also important to note that we focus on Candidates rather than the Program. We do not in this work consider the specifics of identifying all possible candidates of a given program, and we we assume that whatever candidate considered is a possible one for the original program. This translation from program to a set of candidates is something that would be needed in order to practically incorporate our set of conditions in practice while doing transformations.

6 Conclusion and Future work

Our more declarative approach to the ECMAScript memory consistency model results in a relatively compact and concise description of the semantics. This better facilitates mathematical reasoning, which we have used to investigate the conditions on basic optimization operations such as instruction reordering.

Future work is aimed at extending our analysis to validate the conditions for redundancy elimination. We are also interested in further exploring the constraints implied by the potential for multi-byte (non-atomic) accesses. The current standard imposes only very weak conditions on overlapping accesses, but stronger conditions are likely necessary to reflect actual programming practice.

Acknowledgements

We acknowledge the funding in part by the NSERC-COHESA project, under Strategic Networks grant number NETGP485577-15. We would like to thank Conrad Watt for his keen insights and suggestions throughout and Aarti Kashyap for many useful comments and questions that helped more clearly shape the ideas in this paper.

References

1. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer (1996)
2. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL, Austin, TX, USA (2011)
3. Bender, J., Palsberg, J.: A formalization of Java's concurrent access modes. OOPSLA (2019)
4. Draft: ECMAScript language specification (2020), `https://tc39.es/ecma262/#sec-memory-model`
5. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers (1979)
6. Manson, J.: The design and verification of Javas memory model. In: OOPSLA (2002)
7. Naumovich, G., Avrunin, G.S.: A conservative data flow algorithm for detecting all pairs of statement that may happen in parallel. In: FSE, Lake Buena Vista, FL, USA (1998)
8. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-tso. In: TPHOLs, Munich, Germany (2009)
9. Sarkar, S., Sewell, P., Nardelli, F.Z., Owens, S., Ridge, T., Braibant, T., Myreen, M.O., Alglave, J.: The semantics of x86-cc multiprocessor machine code. In: POPL, Savannah, GA, USA (2009)
10. Ševčík, J.: Safe optimisations for shared-memory concurrent programs. In: PLDI, San Jose, USA (2011)

11. Ševčík, J., Aspinall, D.: On validity of program transformations in the Java memory model. In: ECOOP, Paphos, Cyprus (2008)
12. Sewell, P.: Memory, an elusive abstraction. In: ISMM, Toronto, Ontario, Canada (2010)
13. Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., Nardelli, F.Z.: Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In: POPL, Mumbai, India (2015)
14. Watt, C., Pulte, C., Podkopaev, A., Barbier, G., Dolan, S., Flur, S., Pichon-Pharabod, J., Guo, S.: Repairing and mechanising the JavaScript relaxed memory model. In: PLDI, London, UK (2020)