A New Approach to the Semantics of Multithreaded Java

Jeremy Manson and William Pugh

Institute for Advanced Computer Science and Department of Computer Science
University of Maryland, College Park
{jmanson,pugh}@cs.umd.edu

January 13, 2003

Id: newest.tex,v 1.48 2003/01/13 21:17:58 pugh Exp

1 Introduction

The following is a rough overview of the new version of the multithreaded semantics we will propose for Java. This version of the semantics is simplified by assuming that volatiles are fully sequentially consistent; we also don't include anything about final fields, or a number of "corner cases" in the semantics, such as class initialization and treatment of finalizers.

The surface level description of the semantics is very different from our previous description, but the underlying semantics themselves are very similar. The core portion of the old semantics (excluding prescient writes and guaranteed reads) are identical to the core portion of the new semantics. Rather that talking about allWrites, overwritten and previous sets, we just talk about happens-before edges and paths. However, all allWrites, overwritten and previous ever did was to encode happens-before edges.

In our new semantics, it is not necessary for a write to happen before a read for that read to see the value written by that write. This resolves, in a much cleaner way, the same issues and reorderings addressed by prescient writes in our previous semantics.

We have moved away from an operational model for our semantics. Instead, the semantics determine whether a given execution trace for a program is valid.

2 Definitions

Programs A program is a set of threads, each of which is a sequence of statements. A statement exists only in the static definition of a program. We assume simplified statements

that contain at most one thread or heap operation. For example, a statement may read one heap variable, or write one heap variable, but may not increment a heap variable (such a compound statement may be broken down into simplified statements).

Actions Each action corresponds to an activity performed by a dynamic instance of a statement of a program. Example actions include reads or writes of a heap location and locks or unlocks of a monitor. We don't consider compound actions (e.g., incrementing a heap location) or purely thread local actions (e.g., updating a register). Each action has a globally unique identifier, or GUID.

An action is annotated with information about the execution of that action: the variable/monitor accessed, the value read or written, and so on. Reads and writes are also annotated with the variable read or written; this variable is associated with some dynamically determined memory location.

Unithread semantics The unithread semantics are the standard semantics for single threaded programs, and allow the complete prediction of the behavior of a thread based on the values seen by read actions within the thread.

Execution Trace An execution trace (which we sometimes simply call an *execution*) E of a program P is a prefix of an interleaving of the actions derived from the instructions in P that is ordered by the happens-before relationship defined below. Note that this is only a prefix; an execution can end without finishing the program.

There is an important duality here. Sometimes we will refer to executions being ordered solely by happens-before, because that is the only relationship between the actions that is involved in determining what values are seen at any given location. Sometimes we will refer to executions as being an interleaving of actions (implying that they are not solely ordered by happens-before), in order to describe properties of an execution that we cannot easily describe otherwise. In truth, the only ordering that is important in the interleaving is the happens-before ordering.

The actions of an individual thread t within E are consistent with the unithread semantics of t in P. This means that except for read actions of t in E, all of the actions of t in E must be consistent with a standard, unithread execution of t, with each action occurring in the original program order. If a read action in t sees the value of a write by thread t, that write must be the most recent write by t to that memory location. A read action may see a value written by another thread; in that case, the values that can be seen are determined by the memory model.

An execution trace E is a *valid execution trace* if the actions of each thread obey unithread semantics and the values seen by the reads in E are valid according to the memory model.

At the beginning of the execution trace, there is an initial write of the default value (i.e., zero or null) to each variable. This is ordered before the first action of each thread.

When we say that the same action occurs in two different execution traces, we mean that there is an action with the same annotations in each trace (e.g., GUID, variable read and

value seen).

Sequential Consistency An execution trace's results are *sequentially consistent* if they are the same as they would be if the operations of all the threads were executed in some sequential order, the operations of each individual thread appeared in this sequence in the order specified by the program, and there was a happens-before order between each pair of actions in the sequence. In effect, this means that the sequentially consistent results of the program are the ones which could have been produced if all instructions were issued in program order, and the effects of every write were seen in each thread immediately after the write occurred.

The diagram in Figure 6 shows the space of all possible program executions. Clearly, sequentially consistent executions only make up a small portion of the whole. As we describe our memory model, we shall be referring back to this diagram to show how what we are discussing fits with it.

Happens-before edge If we have two actions x and y, $x \xrightarrow{hb} y$ means that x semantically happens before y. Within an execution trace, there is a happens-before edge from each action in a thread t to each following action in t. Furthermore, there is an edge from a unlock action on monitor m to the some subsequent lock action on m, and a happens-before edge from each write to a volatile variable v to some subsequent read of v. There is also a happens-before edge from the initialization of each location to the first action in every thread.

Happens-before path There is a happens-before path $x \xrightarrow{hb} y$ from an action x to a later action y if there is a path of happens-before edges from x to y.

3 Consistent Executions: a first approximation

A memory model is something that can take a program and an execution trace of that program, and tell you whether the execution trace is a legal execution of the program according to the memory model. The memory model works by examining each read in the execution trace and checking that the write observed by that read is allowed.

In this section, we introduce a simple memory model called *consistency*. A partial order is constructed in an execution trace between all of the actions; one action is ordered before another in the partial order if one action happens before the other. We say that a read r of a variable v is *allowed* to observe a write w to v if, in the happens-before partial order of the execution trace:

- r is not ordered before w (i.e., it is not the case that $r \xrightarrow{hb} w$), and
- there is no intervening write w' to v (i.e., no write w' to v such that $w \xrightarrow{hb} w' \xrightarrow{hb} r$).

Initially, a = b = false

Thread 1	Thread 2
b1 = x;	b2 = y;
if (b1)	if (b2)
y = true;	x = true;

x == true, y == false, which is Inconsistent, Cannot Happen

Figure 1: An Inconsistent Execution

Initially, a0: x = 1, y = 0;

Thread 1	Thread 2
a1: r1 = y;	a5: r3 = x;
a2: x = 1;	a6: y = 1;
a3: x = 2;	
a4: r2 = x;	

Figure 2: A Simple Program

In other words, a read r is allowed to observe a write w if, in the partial order of the execution trace, the read is not ordered before the write, and there is no intervening write w' to that location that is ordered between w and r. If all of the reads in an execution are allowed by the consistent memory model, then the execution is consistent. The set of consistent traces of a program P will be written consistent$_P$

As it turns out, this model is too relaxed; it allows executions we need to forbid. However, it does include all of the executions we need to allow, and is worth considering as a first approximation. To keep track of the different models we will discuss, and the behaviors that must be allowed and prohibited, we present Figure 6. The largest circle corresponds to consistency.

An example of an inconsistent execution can be seen in Figure 1. Here, there is no reason for x to see the value true if y sees the value false: since the write to x does not occur, the read should not occur either.

3.1 Simple Example

Consider Figure 2, and the corresponding graph in Figure 3. The solid lines represent happens-before relationships between the actions. The dotted lines between a write and a read indicate a write that that read is allowed to see. For example, the read at `a5` is allowed to see the writes at `a0, a2` or `a3`. An execution is valid under this simple memory model if all reads see writes they are allowed to see. So, for example, the execution that has the result $r1 == 1, r2 == 2, r3 == 1$ is a valid one.

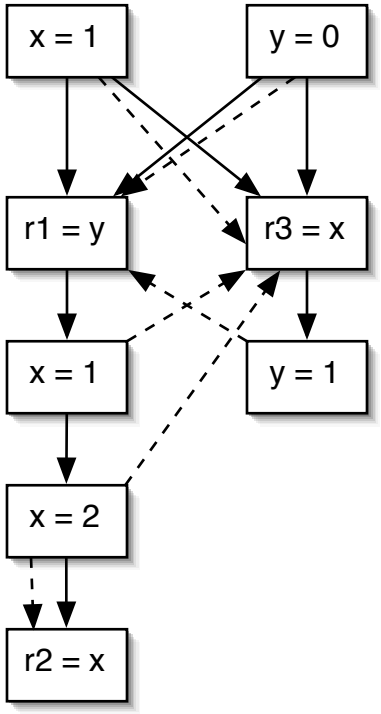

Figure 3: Execution trace of Figure 2

Initially, g0: a = 0; g1: b = 0

Thread 1	Thread 2
g2: x = a;	g4: y = b;
g3: b = 1;	g5: a = 1;

Sequential Consistency Demands $x == y == 1$ Cannot Happen - We Allow It

Figure 4: Sample program - can $x == y == 1$?

3.2 Prescient Write Example

We can start by deriving some valid behaviors of the program in Figure 4. Each statement is labeled with a GUID for the corresponding action.

A compiler may be written to reorder this code so that `g3` occurs before `g2`, and `g5` occurs before `g4`. This might give the result $x == y == 1$. This effect might also be achieved allowing writes to be seen early in the semantics. This example was therefore one of the motivations for prescient writes in an earlier version of the semantics.

There is no ordering between `g3` and `g4`, and no ordering between `g5` and `g2`. Consistency therefore permits `g4` to see the result of `g3`, and `g2` to see the result of `g5`. This provides the same results that prescient writes do: $x == y == 1$.

3.3 Example not Handled by Prescient Writes

So far, we have not seen any examples that could not have been handled by our previous, operational semantics. In this section, we give such an example.

The program in Figure 5 demonstrates one of the limitations of our previous approach. The presence of prescient writes did not allow the result $r1 = r2 = r3 = 1$. However, this behavior must be allowed; if a compiler reordered the read of z to the end of thread 2, then even a sequentially consistent architecture could produce that result. Under our previous semantics, the write of 1 to y could not be guaranteed to occur. It could not, therefore, be performed early. This prevented the result $r1 = r2 = r3 = 1$ from being seen.

The consistent memory model allows us to see $r1 = r2 = r3 = 1$ since there are no orderings between the reads and associated writes. Specifically, there is no happens-before relationship that prevents `g5` from seeing the write in `g3` and `g4` from seeing the write performed by `g8`.

4 Causal Consistency: forbidding causal cycles

Consistency would make a simple definition for execution traces. However, they do have one undesirable trait: consistency allows cyclic dependencies. For example, consider Figure 7, which is a duplication of Figure 1. There is nothing in consistency to prevent the reads of x and y from seeing the value true; those reads would, in fact, be allowed. The result of this trace would appear to come out of thin air.

Initially: g0: x = 0, g1: y = 0, g2: z = 0

Thread 1	Thread 2	Thread 3
g3: x = 1;	g4: r1 = z;	g7: r3 = y;
	g5: r2 = x;	g8: z = r3;
	g6: y = r2;	

Figure 5: Tricky program: Requires $r1 == r2 == r3 == 1$

This is certainly a counterintuitive result, but so might be some of the legal results of Figure 7. The real question is: is this an unreasonable result? What is the difference between a reasonable result and an unreasonable one?

We need a clear way to distinguish between correct programs, which should execute in an intuitive way, and incorrect programs, which can exhibit less obvious behaviors. To do this, we define "correct program" a little more clearly. A program is correctly synchronized if, when you execute it in a sequentially consistent way, there are no data races. The line can now be drawn in a clear way: correctly synchronized programs should behave as if they are sequentially consistent.

Is Figure 7 correctly synchronized? Well, if it is executed in a sequentially consistent way, there are no data races: neither of the writes will occur. Thus, we must forbid the behavior shown in Figure 7.

In our diagram (Figure 6), all of the possible results of this program must fall within the circle marked "sequentially consistent behavior". Since consistent behavior contains executions of this code that are not in that circle, that is insufficient as a memory model.

Another example of this can be seen in Figure 8. The behavior $x == y == 42$ is consistent, as each read each sees a write in the trace, without any intervening happens-before relation.

This program is, of course, incorrectly synchronized, therefore fewer restrictions on its results need to be made. However, we have decided that this behavior is unacceptable in the Java memory model. The value 42 seems to come out of thin air. If we were to allow this, there would be nothing to stop us from allowing references to come out of thin air. This would prevent us from being able to make any guarantees about private and correctly synchronized data remaining private.

We note this example as belonging to a set of "Forbidden out-of-thin-air examples" in Figure 6. We don't, however, attempt to formally define these examples. But there presence is another reason why consistent behavior is too relaxed for use as a Java memory model.

The upshot of this is that we need disallow situations in which an event directly causes itself to happen. In additional to being a desirable property of a memory model, it will also guarantee that correctly synchronized programs exhibit sequentially consistent behavior.

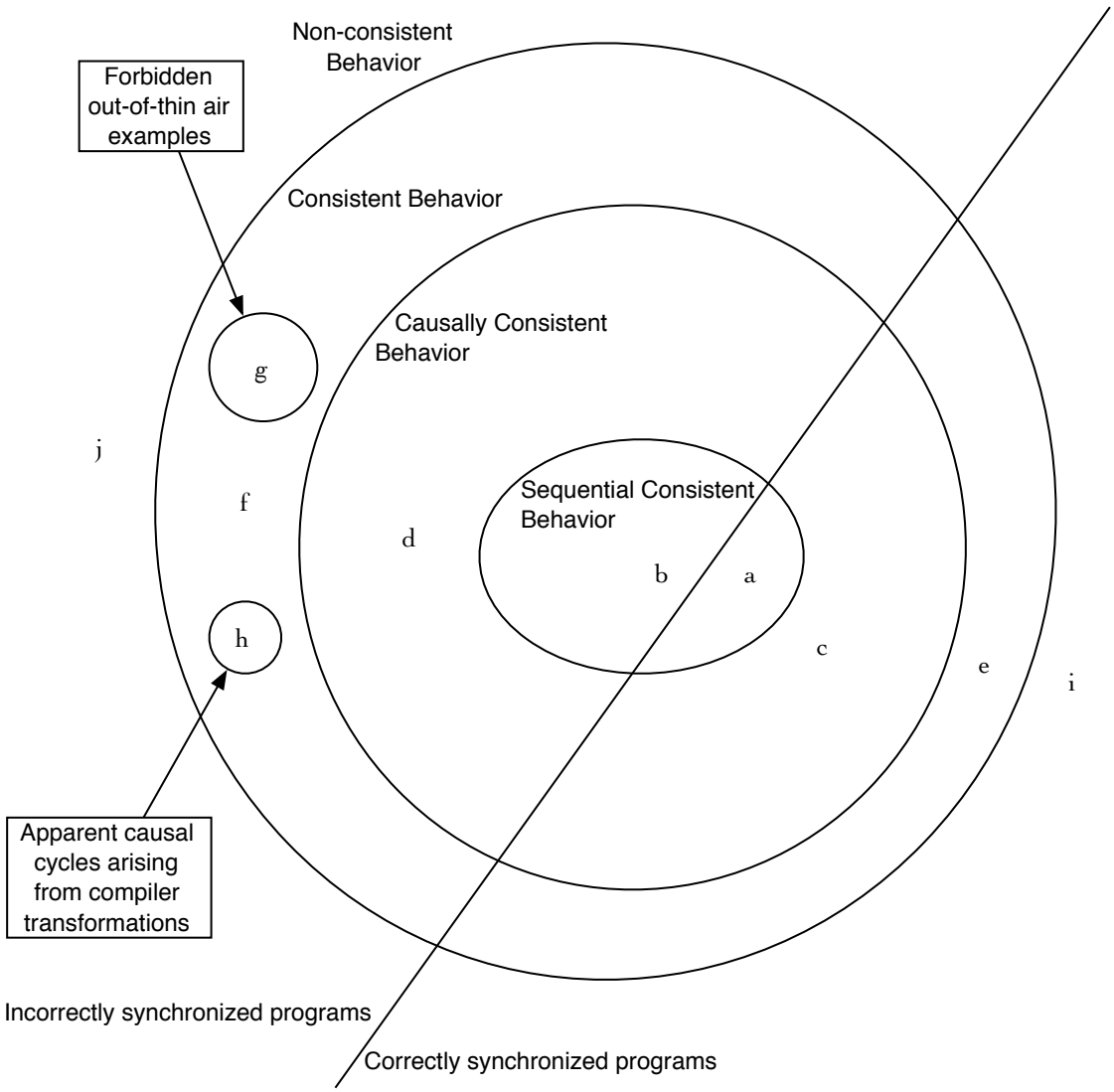

Figure 6: Representation of Space of Executions

Area	Behaviors OK in JMM	Notes	Figure
a	yes		7
b	yes		4
c	no	empty by construction; see Theorem 7.1	n/a
d	yes		5
e	no	must be excluded, so consistent behavior is too relaxed	7
f	yes		
g	no	must be excluded, so consistent behavior is too relaxed	8
h	yes	must be allowed, or important compiler optimizations are forbidden, so causal consistency is too restricted.	12
i	no		1
j	no		

Initially, a = b = false

Thread 1	Thread 2
b1 = x;	b2 = y;
if (b1)	if (b2)
y = true;	x = true;

Correctly Synchronized
x == y == true, which is Consistent, Cannot Happen

Figure 7: Motivation for disallowing some cycles

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
y = r1;	x = r2;

Incorrectly Synchronized: x == y == 42 Cannot Happen

Figure 8: Motivation for disallowing some cycles

4.1 Disallowing causal loops

For any execution trace, we assume the existence of a *causal order*, which is a total order over the actions in that trace. The causal order does *not* have to be consistent with the program order or the happens before order. We use $<_{co}$ to denote the causal order within an execution trace.

The intuition behind causal orders is that for each prefix of that causal order, the next action in the order is *caused* by the actions in the prefix.

Consider an execution trace E of a program P, and an action a in E. Let C be the set of actions in E that occur strictly before a in the causal order of E (in other words, a is not contained in C). Remember that the actions in C do not have to have occurred before a in an execution. We want to say that the set of actions C cause a to occur. Again, the causal order does not have be consistent with the program order, but its results should be determined by the rules for consistent executions .

Unfortunately, *cause* is a rather tricky idea to capture precisely. Informally, we mean that in any legal execution trace E' of P such that C is contained in E', C allows the action a. However, it is circular to define legal execution traces in terms of legal execution traces. This definition is therefore ill-formed, and of little use.

So we have to bootstrap our definition carefully, starting with sequentially consistent executions and building from there in a way that doesn't introduce circular definitions. We start with valid_0 as the set of sequentially consistent executions. We then define valid_{k+1} in terms of valid_k, giving us sets of execution traces $\text{valid}_1, \text{valid}_2, \ldots$.

The process for including a trace E in valid_{k+1} is fairly simple. We take a causal order

$$
\begin{array}{l}
\text{valid}_0 \stackrel{def}{\equiv} \{E \mid E \in \text{sequentially consistent executions of } P\ \} \\
\text{valid}_{k+1} \stackrel{def}{\equiv} \text{valid}_k \cup \{E \mid E \in \text{consistent}_P \wedge \\
\quad \exists co \text{ is a causal order for } E \cdot \\
\quad\quad \forall a : \text{action} \in E \cdot \\
\quad\quad\quad \forall E' : \text{valid}_k \cdot \\
\quad\quad\quad\quad (\{a' \in E \mid a' <_{co}\} \subseteq E') \Rightarrow a \text{ is allowed in } E' \\
\quad\quad\quad \wedge \exists E' : \text{valid}_k \cdot \\
\quad\quad\quad\quad (\{a' \in E \mid a' <_{co}\} \subseteq E') \wedge a \text{ is allowed in } E'\}
\end{array}
$$

E is a valid trace for the memory model if $\exists k \cdot \forall j \geq k \cdot E \in \text{valid}_j$.

Figure 9: Causal Semantics

over E. Each action in E is allowed by a prefix of that causal order. For each action in E, we find the prefix S that allows it, and then examine all of the traces that we have validated that contain S. An action is "okay" if all of the traces that we have validated that contain S allow the action. If all of the actions are "okay", then the execution is a valid one.

The valid sets are bounded by the set of causally consistent executions. This definition is given more formally in Figure 9.

This modification to the semantics disallows the result $x == y == true$ in Figure 7. To justify either of the reads, some legal causal order must contain one of the reads. For a causal order to contain one of the reads, one of the reads must occur in a legal execution. Since we start with sequentially consistent executions, no existing causal order can contain those reads. This is discussed in more detail in Section 7.

4.2 Simple Examples Reexamined

Let us reexamine Figure 4. Our memory model has changed, but we still want the execution trace E that results in $x == y == 1$ to be a legal one. This means that there is a causal order over each of the actions in E that lead to this result.

Consider the causal order { `g3, g4 (sees 1), g5, g2 (sees 1)` }. This is out of program order, but that is allowed; the results are still determined by the rules for consistent executions. The empty trace supports the execution of `g3`. The execution of `g3` supports the read of 1 at `g4`. The execution of { `g3, g4` } allows the write of 1 at `g5` (although it isn't, strictly speaking, necessary: there are no dependencies between `g4` and `g5`). Finally, the execution of { `g3, g4, g5` } allows the read of 1 at `g2`. This causal order therefore supports an execution with the result we want to see.

We also need to reexamine the example in Figure 5. If we are no longer dealing with consistency, we need to validate each action in some trace that results in $r1 = r2 = r3 = 1$. We can do this by using the causal order { `g3, g5, g6, g7, g8, g4` }. Figure 10 demonstrates this more visually.

Trace E_2

	GUID			action	notes
Initialization	g0			x = 0	initial write
Actions	g1			y = 0	initial write
	g2			z = 0	initial write
	T_1	T_2	T_3		
Thread	g3			x = 1	
Actions		g5		r2 = x	sees 1 from g3
		g6		y = r2	writes 1
			g7	r3 = y	reads 1 from g6
			g8	z = r3	writes 1
		g4		r1 = z	sees 1 from g8

Behavior: r1 = 1, r2 = 1 and r3 = 1

Figure 10: Sample Execution Traces for Figure 5

5 Partial Execution Traces

There is an element of execution traces that was introduced in the definition section, but was left unexplained. Recall that an execution trace is only required to contain part of a program: the program need not run to conclusion for the execution trace to be considered.

Although this definition reflects the fact that Java does not provide fairness guarantees, there is a more important reason for its inclusion. Consider Figure 11. Because of its similarity to Figure 4, we know that $x == y == 1$ is a plausible outcome. The twist here is that there is now a third thread, whose control flow is dependent on the actions in Threads 1 and 2.

There are no sequentially consistent results that allow for $x == y == 1$, but that result will get added when those reads are validated in the same execution trace. Therefore, we want `S;` to be able to happen in some execution E. This means that there must be a causal order over the actions in E that occurs in some existing valid_k set, and allows `S;`. If we examine the existing sets valid_k, we will find no traces in which `S;` is allowed, because there are no traces we have legalized yet in which $x == y == 1$ is true.

However, our definition of execution trace includes executions that only cover part of a program. In this case, we simply legalize the execution E' which stops after the reads of x and y in Thread 3. We can now justify the execution of `S;`; because `S;` is allowed in E', it can be part of a legal execution.

6 Full Memory Model

Unfortunately, we have now gone too far in disallowing particular values. We need to allow values that might sometimes seem cyclic to be introduced from compiler optimizations.

Initially, g0: a = 0; g1: b = 0

Thread 1	Thread 2
g2: x = a;	g4: y = b;
g3: b = 1;	g5: a = 1;

Happens before

Thread 3
if (x + y == 2)
S;

Figure 11: Can S; Happen?

Before compiler transformation

Initially, a = 1, b = 0

Thread 1	Thread 2
h1: i = a;	h5: k = b;
h2: j = a;	h6: a = k;
h3: if (i == j)	
h4: b = 2;	

Is i == j == k == 2 possible?

After compiler transformation

Initially, a = 1, b = 0

Thread 1	Thread 2
h4: b = 2;	
h1: i = a;	h5: k = b;
h2: j = i;	h6: a = k;
h3:	

i == j == k == 2 is sequentially consistent

Figure 12: Motivation for allowing some cycles

Consider Figure 12. It would be perfectly reasonable for a compiler to transform the code into Figure **??** by

- eliminate the redundant read of a, replacing h2 with j = i, then
- determine that the expression i == j is now always true, eliminating the conditional branch h3, and finally
- move the write h4: b = 2 early.

Thus, simple compiler optimizations lead to an apparent circular execution trace. Causal consistency allows only the behaviors labeled valid_1 in Figure 14; the behavior $i = j = k = 2$ is forbidden. Defining which circular execution traces we need to allow is, unfortunately, complicated and subtle.

The problem is that there is no causal order that makes $i = j = k = 2$ a causally consistent execution of Figure 14. Consider the causal order { `h4, h5, h6, h1, h2, h3` }. This causal order doesn't work, because the prefix of h4 (b = 2) doesn't guarantee that h4 will occur (no other causal orders work, but we only work through this example). However, we can use the the causal order { `h4, h5, h6, h1, h2, h3` } if we exclude the executions in which i and j see different values, thereby guaranteeing that h4 will execute.

The diagram in Figure 6 shows where some of these executions are: outside of the behavior allowed by Causal Consistency, but still within the boundaries of consistent behavior. The memory model needs to subsume that area; the notion that encapsulates that inclusion is that of *prohibited executions*.

The full semantics can prohibit certain executions based on whether they contain a given read. If the model has a set of valid executions and it wants to prohibit a set of traces prohibited, it must demonstrate that there are alternative valid executions that contain that read. This set, labeled alternativeExecutions, contains tuples $\langle E, r, E' \rangle$, where E is the execution you wish to prohibit, r is the read that defines this prohibited trace, and E' is an alternate execution trace that contains r and the causal trace r has in E, but returns a different value for r. If an execution E is prohibited in alternativeExecutions with an alternate E' listed, E' cannot also be prohibited in alternativeExecutions.

The set of executions allowed under the memory model are the set of valid executions allowed by the semantics with every legal prohibited set. The full semantics for a program P can be seen in Figure 13.

6.1 Cyclic Example - Prohibited Traces

We now have the tools to handle the example given in Figure 12. Remember that we are trying to justify the execution trace E with the result $i == j == k == 2$. We prohibit the traces where the reads of a in Thread 1 return different values. These sets, and the traces that contain them, can be found in Figure 15.

valid_0 contains all of the sequentially consistent traces of this program. Note that these traces do not contain the trace where $i == j == k == 2$, because there is no sequentially

$$\text{valid}_0 \stackrel{def}{\equiv} \{E \mid E \in \text{sequentially consistent executions of } P\ \}$$
$$\text{valid}_{k+1} \stackrel{def}{\equiv} \text{valid}_k - \text{prohibited}_k \cup \{E \mid E \in \text{consistent}_P \wedge$$
$$\exists co \text{ is a causal order for } E \cdot$$
$$\forall a : \text{action} \in E \cdot$$
$$\forall E' : \text{valid}_k - \text{prohibited}_k \cdot$$
$$(\{a' \in E \mid a' <_{co} a\} \subseteq E') \Rightarrow a \text{ is allowed in } E'$$
$$\wedge \exists E' : \text{valid}_k - \text{prohibited}_k \cdot$$
$$(\{a' \in E \mid a' <_{co} a\} \subseteq E') \wedge a \text{ is allowed in } E'\}$$
$$\text{prohibited}_k \stackrel{def}{\equiv} \{E \mid \langle E, r, E' \rangle \in \text{alternativeExecutions} \wedge r \in E' \wedge E' \in \text{valid}_k$$
$$\wedge \text{ let } co \text{ be the causal order used to prove } E' \in \text{valid}_k$$
$$\wedge \{a' \in E' \mid a' <_{co} r\} \subseteq E \wedge r \text{ is allowed but not chosen in } E\}$$

E is a valid trace for the memory model if, for some combination of legal prohibited sets, $\exists k \cdot \forall j \geq k \cdot E \in \text{valid}_j$.

Figure 13: Full Semantics

Behavior					
i	j	k	valid_0	valid_1	notes
0	0	0	Yes	Yes	
0	0	2	Yes	Yes	
0	1	0	Yes	Yes	
1	0	0	No	Yes	
1	1	0	Yes	Yes	
1	1	2	Yes	Yes	
2	2	2	No	No	needs alternative executions to be valid

Figure 14: Consistent Traces for Figure 12

Behavior i	j	k	Prohibited?	An Alternative Behavior i	j	k	valid$_0$	valid$_1$
0	0	0	No	n/a			Yes	Yes
0	0	2	No	n/a			Yes	Yes
0	1	0	Yes	0	0	0	Yes	No
1	0	0	Yes	0	0	0	No	No
1	1	0	No	n/a			Yes	Yes
1	1	2	No	n/a			Yes	Yes
2	2	2	No	n/a			No	Yes

Figure 15: Using alternative executions to allow $i = j = k = 2$

consistent ordering that can express this result. valid$_1$ contains executions whose reads are allowed in the consistent traces in which $i == j$ (because other traces have been prohibited).

Now consider the causal order $S =$ { `h4, h5, h6, h1, h2, h3` }. E is in valid$_1$ if for all actions a in E, there is a prefix of S so that for every $E' \in$ valid$_0$, the presence of S implies that the action is allowed. The first action we pick to validate is `h4`, which happens in every execution in valid$_0$ − prohibited. Then we validate the read of 2 in `h5`, which is allowed in every execution with the causal order { `h4` }. Similarly, the write of 2 in `h6` is allowed in every execution with the causal order { `h4, h5 (reads 2)` }.

Here is the key action: the read of 2 in `h1` is allowed in every execution that contains { `h4, h5 (reads 2), h6` }. This is because of the lack of happens-before orderings in this code.

The read of 2 in `h2` can be validated as well. The inclusion of this last action makes the execution that results in $i == j == k == 2$ legal.

7 Theorems

We say an execution has *sequentially consistent* results if its results are the same as if the actions of all the thread were executed in some sequential order, and the actions of each individual thread appear in this sequence in program order.

Two memory accesses are *conflicting* if they access the same variable and one or both of them are writes. A program is defined to be *correctly synchronized* if in all sequentially consistent executions, any two conflicting accesses are ordered by a happens-before path.

Theorem 7.1 *Consider a correctly synchronized program P. All legal executions of P have sequentially consistent semantics.*

Proof A result is legal in the memory model if, for some combination of legal prohibited sets, $\exists k \cdot \forall j \geq k \cdot E \in$ valid$_j$. If, given a correctly synchronized program, all valid$_k$ contain only

2nd / 1st	Read b	Write b	Lock	Unlock
Read a	yes	$a \neq b$	yes	
Write a	$a \neq b$	$a \neq b$	yes	
Lock				
Unlock	yes	yes		

$a \neq b$: reordering allowed if non-aliased

Figure 16: Reorderings proven valid by Theorem 1

traces which reflect sequentially consistent semantics, then all legal results of this program have sequentially consistent semantics.

For any k, valid$_k$ will contain only sequentially consistent results Proof by induction on k.

Case valid$_0$ valid$_0$ only contains sequentially consistent results for P. Therefore, for a correctly synchronized program, valid$_0$ contains only sequentially consistent results.

Case valid$_{k+1}$ By induction, all execution traces in valid$_k$ are sequentially consistent. Assume $E \in \text{valid}_{k+1} - \text{valid}_k$: E does not have sequentially consistent results. This means that given an ordering over all of the actions in E, there is some read r of a variable v that does not return the value written by the most recent write w to v. Instead, it returns the value of some other write w'.

E can only be in valid$_{k+1}$ if there is a causal order S for E', where $E' \in \text{valid}_k$ and S allows r to see w'. If S allows r to see w', there must be either no happens-before relationship $r \xrightarrow{hb} w$ or no happens-before relationship $w' \xrightarrow{hb} w \xrightarrow{hb} r$. However, note that any two of w, w' and r conflict. Because P is correctly synchronized, there must therefore be a happens-before relationship between w, w' and r in all of its sequentially consistent executions. Since E' is a sequentially consistent execution of P, it must contain a happens-before path between w, w' and r. Therefore, there cannot be a causal order for E' that allows r to see w'. E therefore is not a legal execution of P.

The above is true regardless of the presence of prohibited executions. A prohibited set would only remove sequentially consistent executions between valid$_k$ and valid$_{k+1}$. Since it is still the case that the read will not conflict with the write, which would not make it possible to validate a non-sequentially consistent result.

As a result of this, all valid$_k$ sets for a correctly synchronized program P only contain sequentially consistent results. Therefore all legal results of correctly synchronized programs are sequentially consistent ones.

Theorem 7.2 *Reordering a pair of independent statements as allowed by Figure 16 is legal in the memory model. Figure 16 is not an exhaustive list of the reorderings possible: it is just the reorderings proven valid by this theorem.*

Proof Consider a program P and the program P' that is obtained from P by reordering two adjacent independent statements of a thread t of P as allowed by Figure 16. For example, we can reorder a read and a following lock, but not a read and a following unlock. In saying that statements are independent, we mean that there are no data or control dependences between these statements preventing their reordering.

If every execution E' of P' has the same behavior as some execution E that is valid for P, then all of the possible behaviors of executions of P' are possible behaviors of executions of P. Therefore, the transformation of P to P' will be a legal one.

Consider the case where the reordered instructions consist of a pair of reads and writes (i.e., two reads, a read and a write, or two writes). We want E' to have the same behavior as a valid execution E of P, where $E \in \text{valid}_{k+1}$. E' behaves like an execution in valid_{k+1} if two things are true:

- (i) $E' \in \text{consistent}_P$
- (ii) $\exists co$ is a causal order for $E \cdot$
  $\forall a : \text{action} \in E \cdot$
  $\forall E' : \text{valid}_k - \text{prohibited}_k \cdot$
  $(\{a' \in E \mid a' <_{co} a\} \subseteq E') \Rightarrow a$ is allowed in E'
  $\wedge \exists E' : \text{valid}_k - \text{prohibited}_k \cdot$
  $(\{a' \in E \mid a' <_{co} a\} \subseteq E') \wedge a$ is allowed in E'

$E' \in \text{consistent}_P$ is true if E' obeys the unithread semantics of P and all of the reads in E' are allowed. The actions all occur in their original order, except for the two reordered independent actions. However, the reordering of two independent actions cannot affect the intrathread semantics of the program; therefore, E' obeys the same unithread semantics as E.

We also need to pick causal orders to use to justify the actions in E'. We demonstrate that we can choose causal orders for E' that can be used to justify the actions for some execution $E \in \text{valid}_{k+1}$. The descriptions of these causal orders will therefore imply that all of the same actions take place in E and E'; it is only the positions that have changed.

Causal orders for reorderings of reads and writes For reordered read and write operations, the happens-before relationships in E and in E' are the same. This implies that the causal traces S that allow actions in E will be the same as those in E'.

Causal orders for reorderings of unlock operations with following reads Consider the case of reordering an unlock and a following read. The code motion changes the happens-before relationships from P to P'. In the case of a read r being hoisted above an unlock, the only possible change to the values that can be seen by reads happens because a happens-before relationship $r \xrightarrow{hb} w$ may be introduced between the read and a write w. This would prevent r from seeing the results of that write. E therefore cannot contain a causal order that has r seeing the results of the write.

This is not a problem, however. Since there is no happens-before relationship $w \xrightarrow{hb} r$ in E', it must be the case that r is allowed to see some other write w', where $w' \xrightarrow{hb} r$ and $w' \neq w$. The causal trace justifying execution E will be identical to the one justifying E': the one where r sees the result of w'.

Causal orders for reorderings of writes with following lock operations Consider the description of the causal order for an execution where an unlock operation has been reordered with a following read. The causal trace for E', where the reordering of a write w with a following lock operation has occurred, is similar to that one. Both that reordering and this one deal with a happens-before relationship between a read r and a write w that can occur in the transformed program, but not in the original. In both cases, the read r in E can be forced to see a different write w' that happened before it; this execution provides a valid causal order.

Causal orders for reorderings of unlock operations with following writes Consider the case of reordering an unlock operation with a following write w. The code motion changes the happens-before relationships from P to P'. In the case of a write being hoisted above an unlock, the only possible change to the values that can be seen by reads happens because a happens-before relationship $w \xrightarrow{hb} r$ may be introduced between the write and a read r. This may prevent the read from seeing other writes w'. E therefore cannot contain a causal order that has r seeing the result of w'.

This is not a problem, however. Since there is no happens-before relationship $r \xrightarrow{hb} w$ in E', it must be the case that r is allowed to see the result of w. The causal trace justifying execution E will be identical to the one justifying E': the one where r sees the result of w.

Causal orders for reorderings of reads with following lock operations Consider the description of the causal order for an execution where a lock operation has been reordered with a following write. The causal trace for E', where the reordering of a read r with a following lock operation has occurred, is similar to that one. Both that reordering and this one deal with a happens-before relationship between a write w and a read r that can occur in the transformed program, but not in the original. In both cases, the read r in E can be forced to see the write w; this execution provides a valid causal order.

Because $E \in \text{valid}_{k+1}$, we know that

$$\exists T \text{ is a causal order for } E \cdot \forall a : \text{action} \in E \cdot \exists i : \text{int} \cdot \forall F : \text{valid}_k - \text{prohibited}_k \cdot$$

$$(T[0..i] \subseteq F) \wedge (a \notin T[0..i]) \Rightarrow a \text{ is allowed in } F$$

If T can be used for S to validate the actions in E', then requirement (ii) is fulfilled. Assume that it cannot (i.e., $T \not\equiv S$). There must, therefore, be some action $a \in E'$ such that

$$\neg\exists i : \text{int} \cdot \forall F : \text{valid}_k - \text{prohibited}_k \cdot$$

$$(T[0..i] \subseteq F) \wedge (a \notin T[0..i]) \Rightarrow a \text{ is allowed in } F$$

However, as we have seen, all of the actions in E and E' are the same (only their positions are changed). The happens-before relationships in E and E' are also the same. That means that a must occur in E as well. Therefore, no causal order of T can justify some $a \in E$. This is a contradiction; therefore, $T \equiv S$ must be able to justify the actions in E'.

The truth of conditions (i) and (ii) implies that E' must be in some valid set for P. Because the valid sets are monotonic, it must also be the case that E' is in all valid sets for P, for valid_i, where $i \geq k$. This implies that E' is a legal trace for P. Since all E' are legal traces for P, P' is a legal transformation of P.