What Do High-Level Memory Models Mean for Transactions?

Dan Grossman
University of Washington
djg@cs.washington.edu

Jeremy Manson
Purdue University
jmanson@cs.purdue.edu

William Pugh
University of Maryland, College Park
pugh@cs.umd.edu

Abstract

Many people have proposed adding transactions, or atomic blocks, to type-safe high-level programming languages. However, researchers have not considered the semantics of transactions with respect to a memory model weaker than sequential consistency. The details of such semantics are more subtle than many people realize, and the interaction between compiler transformations and transactions could produce behaviors that many people find surprising. A language's *memory model*, which determines these interactions, must clearly indicate which behaviors are legal, and which are not. These design decisions affect both the idioms that are useful for designing concurrent software and the compiler transformations that are legal within the language.

Cases where semantics are more subtle than people expect include the actual meaning of both strong and weak atomicity; correct idioms for thread safe lazy initialization; compiler transformations of transactions that touch only thread local memory; and whether there is a well-defined notion for transactions that corresponds to the notion of correct and incorrect use of synchronization in Java. Open questions for a high-level memory-model that includes transactions involve both issues of *isolation* and *ordering*.

1. Introduction

1.1 Background and Motivation

With multiprocessors, multi-core architectures and multithreaded programming becoming widespread, shared-memory semantics and synchronization primitives are affecting many more programmers, particularly users of high-level type-safe languages such as Java. Such languages need precise definitions that balance semantic rigor, ease-of-use, and efficient implementation on a variety of available hardware.

Java's shared-memory semantics (i.e., its memory model) [MPA05] is a state-of-the-art example of the importance and difficulty of such definitions. The easy-to-describe memory model of sequential consistency is untenable for modern compilers and runtime environments, even if we assume sequentially consistent hardware (which we cannot); it is debatable whether sequential consistency is easy-to-use, since its availability encourages some programmers to use fragile and error prone concurrency idioms. Equally untenable is making the meaning of incorrectly synchronized code "completely implementation defined," which would sacrifice Java's safety and security guarantees in the presence of data races. Programs with concurrency errors can be likened to programs with buffer overflows: if arbitrary results are allowed, then bugs become less predictable and programs become more vulnerable to external attack.

The Java Memory Model (developed by two of the authors with broad community input), is carefully constructed so that properly synchronized code behaves in a sequentially consistent manner and other code has enough meaning to preserve safety, but not so much as to prevent efficient optimization. Any changes to concurrency in Java (or another language that has well defined multithreaded semantics) must consider its effect on the memory model.

The building blocks of Java synchronization are mutual exclusion locks, condition variables, and non-blocking synchronization operations (via volatile/atomic variables). These mechanisms are pervasive primitives for concurrent programming, and programmers have long been frustrated by their difficulty. They are widely considered difficult to use, with under-use causing races and over-use causing poor performance, even deadlock. Researchers and programming language designers have been constantly on the lookout for new and better ways of describing concurrency, but none has been adopted in a mainstream language.

Recently, researchers (including one of the authors) have proposed complementing or replacing locks in programming languages with a transactional model. Most of the proposed models present a relatively clear model, similar to that of a Conditional Critical Region (CCR) [Hoa02]. Conditional critical regions provide a way to write multiple statements so that they appear to occur *atomically* to the entire system: either all of the CCR is guaranteed to have executed, or none of it will have. On this level, the semantics of these CCRs, or *atomic* blocks, is very simple.

Much work has been done to date in getting transactional solutions to work efficiently and effectively in a programming language [HF03, FQ03, CCM+06, HPST06, ATLM+06, RG05]. However, only cursory attention has been paid to the detailed semantics of transactional concurrency, and how it interacts with the code transformations and optimizations.

In this paper, we raise a litany of difficult questions about how the semantics of atomic blocks interacts with the semantics of a relaxed memory model. Many of these questions arose as we considered different ways to formalize the semantics of transactions, but we believe the important and lasting contribution of this paper is the questions it asks and sample test cases. We believe that any complete semantics of transactions in a programming language must be able to address whether the behaviors described in this paper are allowed. To our knowledge, there has been no prior investigation of these issues (or even acknowledgment of the problem), but resolving them is crucial for making transactions "ready for the real world." Adapting prior work on memory models to incorporate transactions will also raise many interesting intellectual questions.

1.2 Programmers and Implementors

The interaction between the memory model and transactional mechanisms affects two key constituencies. Programmers must know what their programs mean and whether a program is correctly synchronized. Language implementors must know what synchronization barriers must be provided and what compiler optimizations are legal.

Simple answers to relevant questions are often initially appealing but can fail to satisfy either group. For example, consider the seemingly innocuous concept that "all thread-shared mutable memory must be accessed only within transactions." This approach precludes programmers from using idioms generally accepted as correct, such as passing a mutable object between threads via a queue (where synchronization occurs by accessing the queue within transactions, but the passed object can be accessed outside of a transaction).

Such an approach also leaves undefined what the implementation can do if the policy is violated. Under weak atomicity, could a compiler transform

```
atomic{ if (y == 1) x++;}
```

to

```
atomic{ x++; if (y != 1) x--;}
```

Boehm [Boe05] noted this as an issue for compiler transformations and multithreading in C and C++. Allowing this transformation would mean that under weak atomicity, another read could see the value of `x` incremented, despite `y` being not equal to `1`. A language definition must dictate whether such behavior is allowed.

1.3 Key Questions

The specific questions we raise fall into two categories. Questions of *isolation* ask what happens when one thread can observe partial effects of another thread's transaction (if ever) and what constraints (if any) that places on the implementation of transactions. More difficult are questions of *ordering*, which ask what happens when one thread can observe effects of another thread out of order. As a synchronization primitive, we expect the correct use of atomic blocks to reduce such unexpected behavior. (Ideally, we could define such "correct use" and ensure sequential consistency in such a case.) Unfortunately, it is not clear when a pair of transactions should obey the kinds of ordering relationships that Java synchronized blocks using the same monitor do. We present several possibilities and examples, showing none of them are ideal.

2. Isolation

The most essential property of an atomic block is that it appears to execute all-at-once. Informally, that means it must appear to other threads as though the thread executing an atomic block does the entire computation (including all memory reads and writes) at a single point in time. To understand the implications of this on the memory model, we first review the relevant notions of *actions* in the Java Memory Model [MPA05], then extend this notion for both *strong* and *weak* atomicity, and then discuss some unexpected implications.

2.1 Actions in the Java Memory Model

A Java thread executes code in *program order* by performing a sequence of *actions*. For present purposes, we can view these actions as reading memory locations, writing memory locations, and performing synchronization primitives (which we can extend to include starting and committing an atomic block).

In some sense, Java already extends the notion of atomicity to single variables. Every language has some built-in notion of atomicity. For example, at the bit-level, it is impossible to see a partial write — a bit must either be 1 or 0. In Java, writes of 32-bit values are atomic; if one thread writes a 32-bit value, other threads must either see all of the write or none of it. Whether some threads can see the write "before" other threads is a question of ordering, deferred to the next section.

Thread 1	Thread 2
`atomic {`	`x = 1;`
`  x = 0;`	
`  if (x == 1)`	
`    y = 1;`	
`}`	

Can y==1?

Figure 1. Simple example: Strong atomicity cannot see conflicting writes

More formally, reads and writes are atomic actions that are described by (1) the thread executing them, (2) the variable accessed, and (3) a unique identifier. Read actions also include "which write" is observed. For read action r, we write $W(r)$ to denote the unique identifier of the write action that produces the observed value. Implicit in these formal definitions is the atomicity and isolation of accesses to individual variables.

2.2 Strong vs. Weak Atomicity

Following Blundell et al. [BLM05], we distinguish strong atomicity (an atomic block is isolated from all other computation) from weak atomicity (an atomic block is isolated only from other atomic blocks).

To see the difference between strong and weak atomicity informally, consider Figure 1. There is no atomic block protecting the access of `x` in Thread 2. Under strong atomicity, Thread 1 cannot execute `y=1`, since it must appear that no computation occurs between its write of 0 to `x` and its subsequent read. More formally, strong atomicity provides no flexibility on $W(r)$ within a transaction.[1] A key software-engineering advantage of atomicity given this informal definition is that *sequential reasoning* inside an atomic block is sound. For example, we can argue that changing Thread 1 to `atomic{x=0;}` neither adds to nor subtracts from the observable behaviors of the program.

With weak atomicity, Thread 1 is allowed to write 1 to `y`. Whether this could *actually* occur depends on how atomic blocks are implemented (particularly how transaction-local logs are used, if at all). That is, it depends on low-level details that do not belong in the language definition.

To provide a slightly more formal way of characterizing the behavior of strong and weak atomicity, we note that strong atomicity effectively gives three separate guarantees. Each of these guarantees can be removed; a precise definition of weak atomicity must specify which are removed.

The first question asks *whether multiple accesses to the same variable in the same transaction can return the value of different writes, if no write to that variable intervened.* For example, consider Figure 2. Under Java's semantics, if the atomic block were replaced by a synchronized block, the stated result of `r1 == 1, r2 == 2` would be allowed. Strong atomicity does not allow this, because it prevents sequential reasoning – variables cannot be seen to change part of the way through a transaction, because it makes it impossible to reason about the code in isolation.

[1] Transactions with *internal parallelism* (i.e., a transaction that spawns threads that share memory) could reintroduce nondeterminism in $W(r)$.

Initially, x = 0

Thread 1	Thread 2	Thread 3
atomic {	x = 1;	x = 2;
r1 = x;		
r2 = x;		
}		

Can r1 == 1, r2 == 2?

Figure 2. Potential for multiple incoming values in a relaxed memory model

Initially, x = 0

Thread 1	Thread 2
atomic {	r1 = x;
x = 1;	
x = 2;	
}	

Can r1 == 1 ?

Figure 3. Can intermediate values escape an atomic block?

Initially, x = 0

Thread 1	Thread 2
atomic {	r1 = y;
y = 1;	atomic {
if (x == 0)	x = 1;
retry;	}
}	

Can r1 == 1 ?

Figure 4. Is an aborted write visible to another thread?

The second question asks *whether it is possible to see a new value for a variable after it has been written within an atomic block.* This is the question dealt with by Figure 1. Because it also prevents sequential reasoning, strong atomicity disallows it as well.

Instead of addressing which writes might be seen inside the transaction, the final question addresses which writes, performed by the transaction, might be seen by other threads. In short, the question asks *whether an intermediate write, which is later overwritten within the atomic block, can be seen by another thread*. This is the question raised by the example in Figure 3. Strong atomicity also disallows this, because it is atomic with respect to other actions that take place inside the execution.

A closely related question involves whether an intermediate write that is later revoked can be seen by another thread. Consider Figure 4. If the atomic block in Thread 2 is executed first, the write to `y` in Thread 1 is always aborted. Whether it can be seen by Thread 2 or not is a very similar issue to that of whether any intermediate write can be seen by another thread.

In this case, there are two possibilities, each of which depend on the semantic model for an abort. If the semantics of an abort are such that the write in the atomic section occurred, but the abort overwrote it with the original value, then the write can be seen by another thread (as long as the semantics allows other threads to see intermediate writes). If, on the other hand, the semantics of an abort are that any writes the transaction performed never took place, then other threads must be prevented from seeing any updates that took place before the abort occurred.

Ultimately, those who design the semantics of transactions will base this decision heavily on whether intermediate writes are visible to other threads. We therefore do not consider it a separate question.

Initially, x = y = 0

Thread 1	Thread 2	Thread 3
atomic {	r1 = x;	atomic {
x = 1;	y = r1;	r2 = y;
x = 2;		}
}		

Can r2 == 1?

Figure 5. Incorrect code can cause correct transactions to be interleaved

Initially, x = y = 0

Thread 1	Thread 2
atomic {	r1 = x;
x = 1;	r2 = y;
y = 1;	
}	

Can r1 == 1 and r2 == 0 ?

Figure 6. An ordering issue that appears like an isolation issue

2.3 Implications

Having raised three (and a half) questions to define "how weak is weak atomicity," we can consider implications of the answers and problems with even weaker notions.

First, the second and third questions can combine to have implications that may not be immediately obvious and may not have been considered previously. Assume that other threads can see intermediate writes, and consider Figure 5. Can Thread 2 read the value 1 for `x` and, in the same execution, Thread 3 read the value 1 for `y`? If so, then the "incorrect" Thread 2 has the effect of interleaving the two "correct" transactions. This is true even though the two transactions do not access the same memory.

Second, an unsafe language could allow a read outside a transaction that conflicts with writes inside a transaction to lead to arbitrary behavior. In terms of the third question, we could say the read could return any value, even values never written. In terms of Figure 3, this would allow `r1` to be 3. (Section 1 gave a more realistic example of a compiler transformation that could produce such an ephemeral value.) Allowing such behavior in a safe language is unacceptable because it allows incorrectly synchronized code to break the encapsulation and security guarantees of correctly synchronized code.

Finally, we should note that it is easy to confuse questions of isolation with questions of ordering, which we discuss later. For example, in Figure 6, we can have `r1=1` and `r2=0` if we allow reordering the statements in Thread 2 (which is very much in line with the Java Memory Model). Even though the "fix" for this code (assuming such behavior is undesirable) may be to wrap the code in Thread 2 in an atomic block, the reason is ordering – not isolation.

3. Ordering

Ordering determines when actions in one thread can be seen to occur out of order with respect to another. The ordering constraints of a language are determined by its memory model, and are crucial in determining when compilers, runtime environments and hardware architectures can perform code transformations. In this section, we describe some of the decisions it is possible to make about order-

Initially, ready = false, data = 0

Thread 1	Thread 2
atomic {	
ready = true;	r1 = ready;
data = 1;	r2 = data;
}	

Can ready == true, data == 0?

Thread 1	Thread 2
atomic {	
ready = true;	r1 = ready;
data = 1;	if (r1 == true)
}	r2 = data;

Can ready == true, data == 0?

Thread 1	Thread 2
atomic {	
g = o;	r1 = g;
o.x = 1;	r2 = r1.x;
}	

Can r1 != null, r2 == 0?

Figure 7. The reorderings allowed determine which idioms for thread-safe lazy initialization are valid

Thread 1	Thread 2
atomic {	r3 = o;
g = o;	r4 = r3.x;
o.x = 1;	r1 = g;
}	if (r1 != null && r1 == r3)
	r2 = r1.x;

Can r1 != null, r2 == 0?

Figure 8. Compiler optimizations can remove data dependencies

Thread 1	Thread 2
data = 42	tmp = false;
atomic {	atomic {
ready = true;	tmp = ready;
}	}
	if (tmp) {
	r1 = data;
	}

Must r1==42 if tmp == true?

Figure 9. Example of the common idiom of data handoff

ing among atomic blocks, and describe several models that reflect these decisions.

The implications of ordering constraints for atomic blocks, and some of the ways in which they can be subtle, can be seen in Figure 7. In their respective atomic blocks, all three examples set a variable and create a marker indicating that that variable is set. The first two use `data` for the variable, and a boolean variable for the marker, and the last uses an object with a field `x` – if the reference is set, so is the field.

In the leftmost example, it is relatively clear and uncontroversial to say that compilers can reorder the reads in Thread 2. They are not dependent on each other. As a result, `r1` in that example can have the value true, while `data` has the value 0. Note that this does not change the atomicity properties – the atomic block still appears to occur "all at once". It is the ordering of events that has changed.

The second example is slightly more subtle. It does not seem as if the read of `ready` can be reordered with the read of `data`, because there is a control dependence between the two. However, if Thread 2 performed an earlier read of the variable `data`, then it could reuse that value for r2. This has the effect of reordering the two program statements.

The third example is still more subtle. In this case, there is a data dependence between the two statements in Thread 2, instead of a control dependence. You might assume that this couldn't occur because the compiler couldn't possibly read `r1.x` before it knew what `r1` was. But various situations can cause this behavior, such as a read from a stale cache line or address speculation. In some situations, even compiler transformations can perform this apparent impossible reordering. Consider Figure 8, which is the same as the third example, except that additional code has been introduced, as well as an additional conditional test. Here, the compiler would likely replace the read of `r1.x` with a reuse of `r4`.

The distinctions between the different code fragments in Figure 7 may seem subtle, but they are important. They determine the valid idioms for lazy initialization and double checked locking [SH96, BBC+]. The Fortress [ACL+] memory model has been specifically crafted to allow certain kinds of lazy initialization, so that the behavior described in the first example is impossible, but the behaviors described in the second two examples are allowed.

The influence of control and data dependences on allowable reorderings in Java is subtle. In certain cases, they can prevent some questionable program transformations. For a more full treatment, the reader should consult the full memory model [MPA05]. However, the basic building block that allows programmers to control it is *happens-before consistency*. This property can be thought of as a predicate on an execution – if an execution obeys them, then it is a legal execution.

3.1 Accessing Shared Memory Outside of a Transaction

All programming languages need a specific, well-defined way of communicating updates between threads. There are several possible idioms. One interesting idiom, which would be possible in a transactional setting, is to state that all thread-shared mutable memory must be accessed only within transactions. Under this model, the atomic blocks are totally ordered, so that each sees the updates made by the earlier ones.

However, this idiom is unnecessarily limiting. The code in Figure 9 is an example of where a more broad approach might be useful. The variable `ready` is used to indicate that the variable `data` has been initialized. When `ready` is set, the first thread never accesses `data` again. We say that `data` is *handed off* to the second thread. When the second thread accesses `data`, there is no reason for it to incur the overhead of an atomic block if it is aware that no other thread can update it.

In the Java memory model, happens-before consistency is key to defining why this kind of handoff works. Synchronization creates happen-before orderings; if one thread wants to write to a variable that is later read by another thread, those accesses must be ordered by a happens-before ordering. In this section, we discuss how we can use this framework to understand when shared data could be accessed by multiple threads outside of a transaction.

Happens-before consistency in Java uses a happens-before relation to determine what values can be returned by a given read. It uses two orderings:

- *Program Order*, which, for each thread, is a total order over the actions performed in that thread.
- *Synchronization Order*, which is a total order over all synchronization actions in the execution (for the moment, this includes locks and unlocks).

Initially, g=null

Thread 1	Thread 2
`o.x = 1;`	`r1 = g;`
`atomic { }`	`atomic { }`
`g = o;`	`if (r1 != null) {`
	`  r2 = r1.x;`
	`}`

Must r1 != null imply r2 == 1?

Figure 10. Can empty atomic blocks be removed?

There is a happens-before relation between any two actions related by program order. There is also a happens-before relation between an unlock and subsequent lock in the synchronization order, as long as the lock and the unlock are on the same variable. Happens-before order is therefore a partial order over the actions in an execution. We write $a \overset{hb}{\rightarrow} b$ to indicate action a comes before action b in this partial order.

Happens-before consistency says that a read r of a variable v (where a variable is essentially any memory location) is allowed to observe a write w to v if, in the happens-before partial order of the execution:

- r does not happen-before w (i.e., it is not the case that $r \overset{hb}{\rightarrow} w$) – a read cannot see a write that happens-after it, and
- there is no intervening write w' to v, ordered by happens-before (i.e., no write w' to v such that $w \overset{hb}{\rightarrow} w' \overset{hb}{\rightarrow} r$) – the write w is not overwritten along a happens-before path.

Another way of phrasing this would be to say that happens-before consistency implies that it is legal for a read to return the value of a write if that write is not ordered by happens-before with that read (which implies that the read is in a data race with the write), or if it is the last write to the variable in the happens-before order.

The principle of happens-before can be generalized from synchronization to transactions. The most obvious extension of happens-before to transactions involves placing the atomic block, and its entire contents, in the synchronization order, so that there is a total order over all actions protected by atomic blocks. For simplicity, we refer to the total order over these actions as the *atomic order*. It then becomes a simple matter to introduce happens-before edges from the end of each atomic block to the beginning of the next in the total order. Reads can see writes (or not) according to the rules for happens-before consistency.

Returning to Figure 9, if the read of `ready` in Thread 2 occurs later in the atomic order than the write to it in Thread 1, then it will return the value `true`. There will be a happens-before relationship between the writing atomic block and the reading one; since the write to `data` in Thread 1 happens-before the read of it in Thread 2, the read is forced to return the value 42.

3.2 Conflicting Regions

There are subtleties in this definition. For example, happens-before consistency allows redundant happens-before edges to be removed. This means, for example, if a lock is only ever obtained by a single thread, it can be removed. There is no notion of a "lock obtained by a single thread" for an atomic block. However, an implementor might desire to remove an atomic block if a compiler analysis determines that it does not access any shared memory, or is entirely empty.

Consider Figure 10. In this figure, the atomic sections appear to do nothing. However, if we assume that there is a happens-before

Initially, g=null

Thread 1	Thread 2
`o.x = 1;`	`r1 = g;`
`atomic {`	`atomic {`
`  x = 1;`	`  x = 2;`
`}`	`}`
`g = o;`	`if (r1 != null) {`
	`  r2 = r1.x;`
	`}`

Must r1 != null imply r2 == 1?

Figure 11. Can single writes in atomic blocks be no-ops?

relationship between them, then the read in Thread 2 **must** see the value 1, if it occurs. The implementation must insert the proper coherence operations and compiler / memory barriers.

If we wish to remove this constraint on implementations, we must relax our definition of what happens-before edges exist to include less than the full atomic order. In particular, we must have no happens-before edge between these two atomic regions. If there is no happens-before relationship between them, then the read of r1.x can return the value 0. There are several possible definitions of the happens-before relationship that could allow this:

- There is a happens-before relationship between any atomic block that touches shared memory and any subsequent atomic block that touches shared memory.
- There is a way of sorting the various parts of memory into regions. There is a happens-before relationship between any two atomic blocks that touch the same region.
- There is a happens-before relationship between any two atomic blocks that touch the same element in shared memory.

In the memory model literature, when we say that two critical sections "touch" the same piece of memory, we usually mean that both of them access the same memory, and at least one is a write – this is also called a *conflict*. The stricter the definition of "same piece of memory" (whether all of it, some region of it, or a single memory location), the more optimizations possible, but the fewer guarantees provided to programmers.

Note that compilers can remove accesses to shared memory. For example, they can replace the expression `0 * z` with the constant value 0, removing the access to shared memory. However, the happens-before relationships in the original program must be maintained; the compiler can get rid of the accesses to shared memory, but it can't get rid of the compiler or memory barriers that enforce the happens-before relationship. For example, if the implementor uses an atomic machine instruction, such as a compare-and-swap, to ensure that access to a shared variable is guaranteed to be atomic, and the compiler eliminates the access to that variable, then it is vital that some memory synchronization operations be left in to guarantee ordering.

3.3 Happens-Before from Writes to Reads

Another point to make is that even when you focus on atomic blocks that share data, not all idioms require happens-before relationships between all atomic blocks. As mentioned before, in Figure 9, the goal is to "hand off" the variable `data` between threads – Thread 1 initializes it, uses an atomic block to hand it to Thread 2, and then never accesses it again. This idiom only requires a happens-before relationship between the write in Thread 1, and the read in Thread 2.

In fact, this notion generalizes to a wide variety of idioms. It turns out that, for many uses, if an atomic section only contains

Initially, x=0

Thread 1	Thread 2
`atomic {`	`r1 = x;`
`  x = 1;`	`atomic {`
`}`	`  r2 = x;`
	`}`

Can `r2 == 0, r1 == 1`?

Figure 12. Strange behavior allowed by Write $\stackrel{hb}{\rightarrow}$ Read Model

Thread 1	Thread 2	Thread 3
`atomic {`	`atomic {`	`atomic {`
`  x = 1;`	`  x = 2;`	`  r1 = x;`
`}`	`}`	`}`
		`atomic {`
		`  r2 = x;`
		`}`
		`r3 = x;`

Can `r1 == 1, r2 == 2, r3 == 1`?

Figure 13. Another strange behavior allowed by the Write $\stackrel{hb}{\rightarrow}$ Read Model

writes, then it need only form the source of a happens-before edge. Similarly, if an atomic section only contains reads, then it need only form the sink of a happens-before edge. For simplicity, we call this the *write* $\stackrel{hb}{\rightarrow}$ *read model*.

The write $\stackrel{hb}{\rightarrow}$ read model does introduce some changes from the conflicting regions model. The difference can be seen in Figure 11, which is different from Figure 10 only because there are now writes included in the atomic blocks. In the conflicting writes model, there would now be a happens-before relationship between the two, but in the write $\stackrel{hb}{\rightarrow}$ read model, there is now none, so `r1 != null` does not imply that `r2 == 1`.

The write $\stackrel{hb}{\rightarrow}$ read model allows for a fair amount of implementation flexibility. For example, an atomic block that reads from a single variable can be implemented as a volatile read (as defined by the Java memory model), which has very low overhead on most platforms [Lea04].

3.4 Prewrite Ordering

Write $\stackrel{hb}{\rightarrow}$ Read allows some unexpected behaviors. Consider Figure 12. The write $\stackrel{hb}{\rightarrow}$ read model only contains happens-before relationships between atomic blocks that contain writes and subsequent atomic blocks that contain reads. Consider what happens if the atomic block in Thread 2 is ordered before the atomic block in Thread 1. Then, `r2` will contain the value 0. However, there is no constraint on the value of `r1` – it may contain the value 1. This is counterintuitive.

There is a similar issue with the example in Figure 13. Under the write $\stackrel{hb}{\rightarrow}$ read model, there is no ordering between the respective writes – they simply both happen-before the atomic blocks in Thread 3. Neither of them is ever considered to be overwritten by the other. As a result, the read of `x` that assigns to `r3` is free to return either value – 1 or 2 – regardless of whether `r1` and `r2` saw different values.

As far as we know, these results do not result from any useful mixture of program transformations. We can therefore strengthen the write $\stackrel{hb}{\rightarrow}$ read model so that these counterintuitive results are not possible. To do so, we create another ordering, called the *prewrite order*. When an action a is ordered before an action b in the prewrite order, we write $a \stackrel{pw}{\rightarrow} b$.

To strengthen the write $\stackrel{hb}{\rightarrow}$ read model, we use the same rule that there is a happens-before edge between each atomic block that contains a write, and each subsequent atomic block that contains a conflicting read. To this, we add the idea that if there is an atomic block that contains a read or a write, there is a prewrite order edge from that atomic block to any subsequent atomic block that contains a write.

A read r can see a write w unless either $r(\stackrel{hb}{\rightarrow} \cup \stackrel{pw}{\rightarrow})^+ w$, or there is a write w' such that $w(\stackrel{hb}{\rightarrow} \cup \stackrel{pw}{\rightarrow})^+ w' \stackrel{hb}{\rightarrow} r$.

In Figure 12, if the atomic block containing the read occurs before the atomic block containing the write in the atomic order, then there is a prewrite edge from the first to the second. This falls under the rule that a read r cannot see a write w if $r(\stackrel{hb}{\rightarrow} \cup \stackrel{pw}{\rightarrow})^+ w$. The read inside the atomic block in Thread 2 cannot return the value from the write performed by Thread 1.

In Figure 13, there must be a prewrite edge either from the atomic block in Thread 1 to the atomic block in Thread 2, or from the atomic block in Thread 2 to the atomic block in Thread 1. This falls under the rule that a read r cannot see a write w if there is a write w' such that $w(\stackrel{hb}{\rightarrow} \cup \stackrel{pw}{\rightarrow})^+ w' \stackrel{hb}{\rightarrow} r$. Whichever the second write is – Thread 2 or Thread 1 – it will "overwrite" the first. If `r1` is 1, and `r2` is 2, then 1 is overwritten, and `r3` cannot return the value 1.

3.5 A Look at Ordering and Conflicts

This section described design points in the landscape of choices that need to be made for transactional semantics. The happens-before ordering between each atomic block provides the strongest guarantees, followed by the write $\stackrel{hb}{\rightarrow}$ read order with a prewrite order, concluded by the write $\stackrel{hb}{\rightarrow}$ read order without the prewrite order.

Ultimately, though, there is no single answer to the question "which is the right ordering model for my transactional system?". This will depend on the needs of individual runtime designers, and will evolve as the community gets a better sense of what large-scale applications with transactions look like.

4. Related Work

Despite some work defining software transactions and substantial work defining memory models, we do not believe their interaction has been questioned previously.

4.1 Semantics of Transactions

No work we are aware of has considered compiler transformations or a relaxed memory model when defining the meaning of atomic blocks.

Blundell et al. [BLM05] coined the terms *strong* and *weak* to distinguish whether non-transactional code could examine mid-transaction state. They showed that neither semantics subsumes the other (e.g., each allows some programs to terminate that the other does not). They did not consider ordering issues, which we have demonstrated are largely orthogonal and thornier.

Harris et al. [HMJH05] present a high-level formal operational semantics describing Haskell's composable transactions. In this idealized semantics, only one thread executes at a time, which precludes considering a memory model weaker than sequential consistency.

Vitek et al [VJWH04] extend the Featherweight Java formalism [IPW01] such that different implementations of transactions can be formalized by redefining certain operations, such as *write* or

commit. Correctness of such an implementation is defined in terms of serializability. While the model supports keeping changes invisible to other threads until a transaction commits, it has no notion of viewing changes out of order as in a relaxed memory model.

Scott [Sco06] presents a framework for transactional semantics that makes inconsistent reads and writes explicit (requiring a correct implementation to abort transactions performing such operations). This detail allows definitions regarding when a transaction must succeed, an issue we have not considered. Conversely, Scott considers only a total order on all memory operations, i.e., sequential consistency.

To our knowledge, work on hardware transactional memory has also not considered interaction with the hardware memory model (and, of course, doing so would still leave unanswered questions regarding high-level compiler transformations). The `tcc` project [HCW+04] supported transactions that had to commit in order, but this feature was designed for speculative parallelization of sequential loops. More recent work [MCC+06] has focused on an expressive instruction-set architecture without explicit mention of the memory model.

4.2 Memory Models

Hardware architectures have motivated most prior work on memory models. Lamport provides one of the earliest discussions of memory models [Lam78], which provides the widely used definition for sequential consistency. Several relaxed models have been proposed in academia and for commercial hardware; Adve and Gharachorloo provide a primer for this work [AG96]. Adve and Hill [AH90, AH93] and Gharachorloo et al. [GLL+90] first formalized the property of sequential consistency for data-race-free (or properly-labeled [GLL+90]) programs for memory models. None of the above work takes transactions into consideration.

Our work is primarily motivated by requirements for programming languages, which differ significantly from those for hardware-driven memory models. There have been many other attempts to write models for Java (e.g., [AMS00], [YGL01]). In programming languages such as C++ [Str97], threads are not part of the language specification. Instead, libraries such as POSIX threads (pthreads) [LB98] support multithreading. Recent work has started towards a unified memory model for C++ [Boe05].

To our knowledge, the issues we address with transactions are not ones that have been addressed before. Harris and Fraser [HF03] described a simple memory model for their transactional memory system that followed the Java memory model by inducing an ordering between atomic blocks that access the same memory locations. However, their work did not describe the implications of these choices in the detail we have provided here, especially with regard to program transformation. It also did not describe alternative techniques.

Welc et al. [WJH05] provided a technique for aborting and retrying synchronized blocks that dealt with happens-before issues, but they were interested in preserving Java semantics for locks, rather than defining new semantics for transactions.

5. Conclusion

We have considered several questions of shared-memory semantics that must be answered before software transactions can be added to a high-level language with a relaxed memory model. Such a memory model is necessary for conventional compiler optimizations and execution on modern hardware, so these issues appear unavoidable. We hope these questions provide a timely springboard for community-wide discussion and consensus, so that the promise of transactions is not lost due to each implementation providing its own ill-defined semantics. As it stands currently, straightforward implementations of transactional mechanisms may violate existing programming language memory models.

To move forward, it may help to classify relevant questions into three levels of current understanding. First, there are well-understood questions for which the community can hopefully agree on answers. Examples include the "how weak is weak atomicity" questions of isolation and whether there are ordering constraints between transactions that access only thread-local data. Second, there are reasonably clear concepts that need some refinement. One example is how to support a definition of ordering constraints based on "accessed memory" when a compiler may change what memory is accessed. Finally, there are implications (both good and bad) we have not yet considered. For example, when using synchronized blocks, there are easily defined idioms sufficient for sequentially consistent behavior. Are there analogous idioms for transactions and is it reasonable to exhort programmers to follow them?

References

[ACL+] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification, version 0.903. http://research.sun.com/projects/plrg/fortress0903.pdf.

[AG96] Sarita Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[AH90] Sarita Adve and Mark Hill. Weak ordering—A new definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 2–14, 1990.

[AH93] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[AMS00] Arvind, Jan-Willem Maessen, and Xiaowei Shen. Improving the Java memory model using CRF. In *OOPSLA*, pages 1–12, October 2000.

[ATLM+06] Ali-Reza Adl-Tabatabai, Brian Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *ACM Conference on Programming Language Design and Implementation*, pages 26–37, 2006.

[BBC+] David Bacon Joshua Bloch, Jeff Bogda, Cliff Click, Paul Haahr, Doug Lea, Tom May, Jan-Willem Maessen, John D. Mitchell Kelvin Nilsen, Bill Pugh, and Emin Gun Sirer. The double-checked locking is broken declaration. http://www.cs.umd.edu/ pugh/java/memoryModel/DoubleCheckedLocking.ht

[BLM05] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Proceedings of the 2005 Workshop on Duplicating, Deconstructing and Debunking*, 2005.

[Boe05] Hans-J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 261–268, New York, NY, USA, 2005. ACM Press.

[CCM+06] Brian D. Carlstrom, JaeWoong Chung, Austen McDonald, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. The Atomos transactional programming language. In *ACM Conference on Programming Language Design and Implementation*, pages 1–13, 2006.

[FQ03] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *ACM Conference on Programming Language Design and Implementation*, pages 338–349, June 2003.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy.

Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Ann. Intl. Symp. on Computer Architecture*, pages 15–26, May 1990.

[HCW+04] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (TCC). In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–13, 2004.

[HF03] Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 388–402, Seattle, Washington, November 2003.

[HMJH05] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2005.

[Hoa02] C. A. R. Hoare. Towards a Theory of Parallel Programming. *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, pages 231–244, 2002.

[HPST06] Timothy Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *ACM Conference on Programming Language Design and Implementation*, pages 14–25, 2006.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, 1978.

[LB98] Bil Lewis and Daniel J. Berg. *Multithreaded programming with pthreads*. 1998.

[Lea04] Doug Lea. JSR-133 Cookbook, 2004. Available from http: //gee.cs.oswego.edu/dl/jmm/cookbook.html.

[MCC+06] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *33rd International Symposium on Computer Architecture*, June 2006.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM Press.

[RG05] Michael F. Ringenburg and Dan Grossman. AtomCaml: First-class atomicity via rollback. In *10th ACM International Conference on Functional Programming*, pages 92–104, September 2005.

[Sco06] Michael L. Scott. Sequential specification of transactional memory semantics. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

[SH96] Douglas Schmidt and Tim Harrison. Double-Checked Locking: An Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects. In *Third Annual Pattern Languages of Program Design Conference*, 1996.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman, Reading Mass. USA, 3rd edition, 1997.

[VJWH04] Jan Vitek, Suresh Jagannathan, Adam Welc, and Antony L. Hosking. A semantic framework for designer transactions. In *13th European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 249–263, 2004.

[WJH05] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Revocation Techniques for Java Concurrency. *Concurrency and Computation:Practice and Experience*, 2005.

[YGL01] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Specifying Java thread semantics using a uniform memory model. In *ACM Java Grande Conference*, November 2001.