Models of Memory Consistency

Cache coherence ensures that multiple processors see a consistent view of memory. It does not answer the question of *how* consistent the view of memory must be. By ":how consistent" we mean, when must a processor see a value that has been updated by another processor? Since processors communicate through shared variables (used both for data values and for synchronization), the question boils down to this: In what order must a processor observe the data writes of another processor? Since the only way to "observe the writes of another processor" is through reads, the question becomes, what properties must be enforced among reads and writes to different locations by different processors?

Although the question of how consistent memory be seems simple, it is remarkably complicated, as we can see with a simple example. Here are two code segments from processes P1 and P2, shown side by side:

```
P1: A = 0;              P2: B = 0;
    .....                   .....
    A = 1;                  B = 1;
L1: if (B == 0) ...     L2: if (A == 0)...
```

Assume that the processes are running on different processors, and that locations `A` and `B` are originally cached by both processors with the initial value of 0. If writes always take immediate effect and are immediately seen by other processors, it will be impossible for *both* if-statements (labeled `L1` and `L2`) to evaluate their conditions as true, since reaching the if-statement means that either `A` or `B` must have been assigned the value 1. But suppose the write invalidate is delayed, and the processor is allowed to continue during this delay; then it is possible that both P1 and P2 have not seen the invalidations for `B` and `A` (respectively) *before* they attempt to read the values. The most straightforward model for memory consistency is called ***sequential consistency***. Sequential consistency requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved. Sequential consistency eliminates the possibility of some nonobvious execution in the previous example, because the assignments must be completed before the if statements are initiated.

The simplest way to implement sequential consistency is to require a processor to delay the completion of any memory access until all the invalidations caused by that access are completed. Of course, it is equally effective to delay the next memory access until the previous one is completed. Remember that memory consistency involves operations among

different variables: the two accesses that must be ordered are actually to different memory locations. In our example, we must delay the read of `A` or `B` (`A==0` or `B==0`) until the previous write has completed (`B=1` or `A=1`). Under sequential consistency, we cannot, for example, simply place the write in a write buffer and continue with the read.

Relaxed Consistency Models:

The key idea in relaxed consistency models is to allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering, so that a synchronized program behaves as if the processor were sequentially consistent. There are a variety of relaxed models that are classified according to what orderings they relax. The three major sets of orderings that are relaxed are:

1. The W→R ordering: which yields a model known as total store ordering or processor consistency. Because this ordering retains ordering among writes, many programs that operate under sequential consistency operate under this model, without additional synchronization.
2. The W→W ordering: which yields a model known as partial store order.
3. The R→W and R→R orderings: which yields a variety of models including weak ordering, the Alpha consistency model, the PowerPC consistency model, and release consistency depending on the details of the ordering restrictions and how synchronization operations enforce ordering.

By relaxing these orderings, the processor can possibly obtain significant performance advantages. There are, however, many complexities in describing relaxed consistency models, including the advantages and complexities of relaxing different orders, defining precisely what it means for a write to complete, and deciding when processors can see values that the processor itself has written.