March 16, 2004, 9:03am

Semantics allows reordering

Theorem 1 *Consider a program P and the program P' that is obtained from P by reordering two adjacent statements s_x and s_y. Let s_x be the statement that comes before s_y in P, and after s_y in P'. The statements s_x and s_y may be any two statements such that*

- *reordering s_x and s_y doesn't eliminate any transitive happens-before edges in any valid execution (it will reverse the direct happens-before edge between the actions generated by s_x and s_y),*
- *s_x and s_y are not conflicting accesses to the same variable,*
- *s_x and s_y are not both synchronization actions, and*
- *the intra-thread semantics of s_x and s_y allow reordering (e.g., s_x doesn't store into a register that is read by s_y). This means that common single-threaded compiler optimizations are legal here.*

Transforming P into P' is a legal program transformation.

Proof: Assume that we have a valid execution E' of program P'. To show that the transformation of P into P' is legal, we need to show that there is a valid execution E of P that has the same observable behavior as E'.

Since E' is legal, we have a sequence of executions, $E'_0, E'_1, \ldots E'_n$, with $E'_n \equiv E'$ such that E'_0 doesn't have any committed actions and E'_i is used to justify the additional actions that are committed to give E'_{i+1}.

We will show that we can use $E_i \equiv E'_i$ to show that $E \equiv E'$ is a legal execution of P.

Let x be the action generated by s_x and y be the action generated by s_y.

If x and y are both uncommitted in E'_i, the happens-before ordering between x and y doesn't make any difference in terms of the possible behaviors of actions in E_i and E'_i. Any action that happens-before x or y happens-before both of them. and if either x or y happens-before an action, both of them do (excepting, of course, x and y themselves). Thus, the reordering of x and y can't effect the write seen by any uncommitted read.

Similarly, the reordering doesn't effect which incorrectly synchronized write a read can be made to see when the read is committed.

If E'_i is used to justify committing x in E'_{i+1}, then if E_i is used to justify committing x in E_{i+1}. Similarly for y.

If one or both of x or y is committed in E'_i, it can also be committed in E_i, without behaving any differently, with one caveat. If y is a lock or a volatile read, it is possible that committing x in E'_i will force some synchronization actions that happen-before y to be committed in E'_i. However, those actions can be committed in E_i even if we are not forced to do so.

Thus, the sequences of executions used to justify E' will also justify E, and the program transformation is legal. □

Correctly Synchronized Programs exhibit only SC Behaviors

We say an execution has *sequentially consistent* (SC) behavior if there is a total order over all actions consistent with the program order of each thread such that each read returns the value of the most recent write to the same variable. Two memory accesses are *conflicting* if they access the same variable and one or both of them are writes.

Definition 1 *A program is* **correctly synchronized** *if and only if in all sequentially consistent executions, all conflicting accesses to non-volatile variables are ordered by happens-before edges.*

Lemma 2 *Any execution E of a correctly synchronized program P that is happens-before consistent and in which each read sees a write that happens-before it has sequentially consistent behavior.*

Proof: A topological sort of the happens-before edges of the actions gives a total order consistent with program order and synchronization order. Let r be the first read in E that doesn't see the most recent conflicting write w but instead sees w'. Let E be $\alpha w' \beta w \gamma r \delta$.

Let E' be an execution $\alpha w' \beta w \gamma r' \delta'$, obtained by performing the actions $\alpha w' \beta w \gamma$, then the action r', which is the same as r except that it sees w, and δ' is any sequentially consistent completion of the program, such that each read sees the previous conflicting write.

The execution E' is sequentially consistent, and it is not the case that $w' \stackrel{hb}{\rightarrow} w$, so P is not correctly synchronized.

Thus, no such r exists and the program has sequentially consistent behavior. □

Theorem 3 *Any execution E' of a correctly synchronized program is sequentially consistent.*

Proof: By lemma 2, if an execution E' is not sequentially consistent, there must be a read r that sees a write w such that w does not happen-before r. Furthermore, the read must be committed. There may be multiple such reads; if so, let r the first such read that was committed. Let E be the execution that was used to justify committing r.

Our notion of correspondence requires that the relative happens-before order of the committed and actions being committed remain the same in all executions using the resulting set of committed actions. Thus, if we don't have $w \stackrel{hb}{\rightarrow} r$ in E', then we didn't have $w \stackrel{hb}{\rightarrow} r$ in E when we committed r.

Since r was the first read to be committed that didn't see a write that happens-before it, each committed read in E sees a write that happens-before it. A non-committed read always sees a write that happens-before it. Thus, each read in E sees a write that happens-before it, and there is a write w in E that is not ordered with respect to r by happens-before ordering.

A topological sort of the actions in E according to happens-before edges of the actions gives a total order consistent with program order and synchronization order. This gives a sequentially consistent execution in which the conflicting accesses w and r are not ordering by happens-before edges, and thus the program is not correctly synchronized. □