February 6, 2004, 1:46pm

Proof sketch that Manson/Pugh allows reordering

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

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

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'.

Assume $E' = \langle S, so, hb', jo \rangle$. We are going to show that $E = \langle S, so, hb, jo \rangle$ is also a valid execution of P. Let a_x and a_y denote the actions corresponding to the statements x and y. Because of the reordering the happens-before ordering may be different but we know that $hb - \{a_x \rightarrow a_y\} \subseteq hb' - \{a_y \rightarrow a_x\}$. Clearly, if E' is consistent then E is consistent, so we only need to worry about showing that jo is justified as the justification order of E.

- Assume that $jo = \alpha a_y \beta a_x \gamma$.

  We don't need to worry about any actions that were prescient in E'. The justification of those prescient actions in E' will also justify them in E.

  The only action that could be prescient in E but not E' is a_y. If a_y is not prescient in E', we know a_x is the only action that comes after a_y in the justification order such that $a_x \stackrel{hb}{\rightarrow} a_y$. In order to justify the prescient action a_y in E, we need to show that in each non-forbidden NPE of α, an action a'_y congruent to a_y occurs.

  - If a_y is a read, then in order for a'_y to be congruent to a_y we must show that the write seen by a_y in E is visible to a'_y.
  - If a_y is a write, we need to show that the Prescient Write Rule allows a_y to occur presciently.

  We know intra-thread semantics will cause a_y to occur, since all actions other than a_x that occur before a_y in program order are in α, and we have assumed as a condition for the reordering that a_x does not affect the intra-thread semantics of a_y.

  In any non-prescient extension E'' of α, the only actions that can happen-before a_y are either in α or the action corresponding to a_x. This is because in E', the action a_y is

non-prescient, so all actions that happen-before a_y are in α. In E, due to the ordering of the statements that generate a_x and a_y, a_x is also ordered before a_y, but since a_x is not an acquire action, it doesn't induce any happens-before edges that cause any other actions to happen-before a_y.

If a_y is a read that sees a write w in α, then the writes that can be seen by a'_y are determined by the conflicting writes that occur before a'_y in the justification order and by the conflicting writes that happen-before a'_y. Since these are the same in both E and E', a'_y will also be able to see w in E'.

If a_y is a write, we also need to prove that the Prescient Write Rule allows a_y to occur presciently. But in all non-prescient extensions of α, all conflicting reads r such that $r \xrightarrow{hb} a_y$ are in α, so the number of such reads by each thread is the same in both the justifying execution and the execution being justified.

- Alternatively, assume that $jo = \alpha a_x \beta a_y \gamma$. Then any action in E that is prescient is also prescient in E', and the justifications used to show that those actions are justified in E' will also show that those actions are justified in E.

Proof Sketch that Model Allows Unrolling / Merging

Compiler transformations can take place that take code that executes along one control path, and split that path so it executes along multiple control paths that are equivalent to the original. Conversely, it can take code that executes along multiple control paths, and merge these paths so it only executes along one control path.

Consider a program P, and a program P' that is generated by splitting or merging. Is it the case that every execution E' of P' has a corresponding execution E in P?

All forms of splitting and merging control paths must preserve intra-thread semantics. It is therefore only the inter-thread actions that may be affected by splitting and merging. Because such actions do not need to correspond to program statements, other than that they must obey the intra-thread semantics of the program, any execution E' of P' will have the same actions of an execution E of P.

Proof Sketch that Model Allows Speculative Reads

Some systems perform speculative reads. This proof describes certain kinds of speculative reads, and shows that they are allowed by the memory model. The proof doesn't say anything about other kind of speculative reads (they may very well be allowed by the memory model, but that fact is not shown in this proof).

A speculative read is one that is executed before it is known if the read will occur or what address will be read. If the speculation is wrong, the read is invalid, and anything dependent on the read must be re-performed at the appropriate place. If it is found to be invalid when the read was supposed to occur, the read is performed again, where it was originally supposed to be performed. A speculative read cannot occur earlier than the last synchronization action that performed an acquire, or earlier than a write to the variable from

which it reads. We will call the early read the *speculation point*, and the original location of the read will be the *original point*.

The value read at the speculation point must be legal to read at the original point under the memory model. A read must see a value that is written before that read in the justification order. If the justification order for an execution where the read occurs at the speculation point is $E' = \alpha r \beta \gamma$, then an equivalent execution where the read occurs at the original point would be $E = \alpha \beta r \gamma$, where r sees a value written in α. Assume that the value that r returned in E' could not have been returned in E. Then either

- The variable was re-written between the speculation point and the original read point. In this case, the read was invalid and would have been re-performed (by definition).
- The variable was written by another thread, and this thread performed an acquire that forced it not to see the value read. Since there are no acquires between the speculative point and the original point, this, too, is impossible.

Finally, the speculative read cannot influence its own validity, because its return value is not used until the original point; the validity is determined before this.

Correctly Synchronized Programs exhibit only SC Behaviors

We say an execution has *sequentially consistent* (SC) 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* (CS) if in all sequentially consistent executions, any two conflicting accesses are ordered by a happens-before path.

A justification order will return a non-SC result if a read returns a value of a write that does not happen before that read.

Lemma 1 *If an execution E has a non-prescient justification order and all conflicting accesses are ordered by happens-before edges, E has sequentially consistent behavior.*

Proof: Assume we have an execution E with a non-prescient justification order jo. Since jo is non-prescient, the ordering of the actions in jo is an valid sequentially consistent execution order. The only way the execution could not be sequentially consistent is if a read of a variable v, rather than seeing the most recent write to v, sees an earlier write to v. But all of those accesses are ordered by happens before edges, so only the most recent write to v is visible. So only sequentially consistent behaviors are allowed. □

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 *In all non-prescient executions of correctly synchronized programs, all conflicting accesses are ordered by happens-before edges.*

Proof: By contradiction. Assume there is a non-prescient execution with a justification order $\alpha x \beta$, where x is the first action on the justification order that is not correctly synchronized with respect to conflicting actions that occur before it in the justification order.

Let x' be an action congruent to x; if x is a read of a variable v, then x' sees the last write to v in α.

By executing additional actions β' according to sequentially consistent semantics, we arrive at a sequentially consistent execution with a justification order $\alpha x' \beta'$ that has a data race. So this program must not have been correctly synchronized. □

Lemma 3 *All non-prescient executions of correctly synchronized programs exhibit sequentially consistent behavior.*

Proof: Follows directly from Lemmas 2 and 1.

Definition 2 *Let E be an execution with a justification order $\alpha x y \beta$ such that x is prescient, y is not prescient, and y is not a read that sees x and x and y are not both synchronization actions. Then $\alpha y x \beta$ is the justification order of an execution E' that is the* **prescient relaxation** *of x in E. Note that E and E' have the same actions, behavior, synchronization order and happens-before edges.*

Definition 3 *Given any execution E, the* **full prescient relaxation** *of E is an execution E' that is obtained by applying the reordering described in Definition 2 repeatedly until no more reorderings are possible.*

Lemma 4 *For any prefix α of a justification order of an execution of a correctly synchronized program, full prescient relaxation of any non-prescient extension of α gives an equivalent execution with no prescient actions.*

Proof: Our proof is by induction; the inductive hypothesis is that for every execution E that is a non-prescient extension of α (i.e., $E \in NPE(\alpha)$), full prescient relaxation of E gives an execution with no prescient actions.

Base Case. In the base case, α is the empty prefix. All non-prescient extensions of the empty prefix are non-prescient. □

Inductive Case. Given the inductive hypothesis, we must show that for every execution $E \in NPE(\alpha x)$, the full prescient relaxation of E is entirely non-prescient.

Every $E \in NPE(\alpha x)$ has a justification order $\alpha x \beta$, where β is non-prescient. Repeatedly apply prescient relaxation to x in E.

- If this makes x non-prescient, then the resulting execution has a justification order that is a non-prescient extension of α, and our inductive hypothesis tells us that full prescient relaxation of E gives an entirely non-prescient execution.

- Alternatively, after relaxing x zero or more times, we have a justification order $\alpha\beta' xy\gamma$, where x is prescient and cannot be further relaxed, and y, β' and γ are non-prescient.

  - It cannot be the case that both x and y are synchronization actions, since then the justification order of E wouldn't respect the synchronization order of E.
  - Then it must be the case that y is a read r that sees the write x.

    If $x \xrightarrow{hb} r$, then r would also be prescient. So we know there is no happens-before edge from x to r. Since r sees x, we also known that there is no happens-before edge from r to x.

    The action x does not influence whether r occurs or the variable it reads; it only influences the value it sees. We can extend $\alpha\beta'$ with a non-prescient read r' corresponding to r (seeing a write in $\alpha\beta'$). Every non-forbidden non-prescient extension of $\alpha\beta' r'$ must include an x' that corresponds to x; otherwise, it would not be possible to include x as the next prescient action in the justification order after $\alpha\beta'$. Let E' be any of those non-forbidden, non-prescient extensions.

    Let E'' be the full prescient relaxation of E'. Because $\beta' r'$ is non-prescient, we know that E' is a non-prescient extension of α. Therefore, by the inductive hypothesis, E'' must be entirely non-prescient. By Lemma 2, all conflicting accesses are ordered by happens-before edges in E''. Since prescient relaxation does not change actions or happens-before edges, all conflicting actions in E' must also be ordered by happens-before edges. So we know that $r' \xrightarrow{hb'} x'$ in E'.

    Let t be the thread that performs both r' and r. Let c be the number of conflicting reads in E' performed by t that happen-before x'. Since $r' \xrightarrow{hb'} x'$ in E' and r' is

not in $\alpha\beta'$, we know that c is strictly greater than the number of conflicting reads performed by t in $\alpha\beta'$.

By the prescient write rule, we know that there must exist at least c conflicting reads performed by t in $\alpha\beta' x r \gamma$ that happen-before x. Since c is greater than the number of conflicting reads in $\alpha\beta'$, we know that there must be at least one such read in $r\gamma$. Let r'' denote some such read.

So this gives us three possibilities:

* r'' is r. But since $r'' \xrightarrow{hb} x \wedge \neg(r \xrightarrow{hb} x)$, this is a contradiction.
* $r \xrightarrow{po} r''$. But if $r \xrightarrow{po} r'' \xrightarrow{hb} x$, then $r \xrightarrow{hb} x$, which is a contradiction.
* $r'' \xrightarrow{po} r$. But since r'' in γ, this would mean that r is prescient, which contradicts our assumption that r is non-prescient.

Therefore, our inductive hypothesis holds. □

Theorem 5 *Correctly synchronized programs have sequentially consistent semantics.*

Proof: All executions of correctly synchronized programs are equivalent to non-prescient executions of the same program because of Lemma 4. Therefore, by Lemma 3, all executions of correctly synchronized programs exhibit sequentially consistent behavior. □