January 21, 2004, 1:51pm

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 $E' = \langle S, so, hb', co' \rangle$. We are going to show that $E = \langle S, so, hb, co \rangle$ 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 a co that is valid as the justification order of E.

- Assume that $co' = \alpha a_y \beta a_x \gamma$, and that a_y is a read. Then $co = \alpha \beta a_x a_y \gamma$ is a valid justification order for E. Note that the happens-before ordering and the write seen by each read are identical.

  Since the happens before edges are the same and the write seen by a_y in the execution E' is in α, a_y can see that same write in E (because it is happens-before consistent and the write comes before the read in the justification order).

  Additionally, we know that a_y does not affect any of the actions in β, because it is a read that is not written out before β occurs in co'. It doesn't affect a_x by assumption. It does not affect what values can be read in γ because it does not introduce additional happens-before relationships with any actions in γ.

  Since a_y can see the same value in co that it does in co', and its seeing that value does not affect any other values seen by co, co is a valid justification order for E.

- Alternatively, assume that $co' = \alpha a_y \beta a_x \gamma$, and that a_y is not a read. We will show that we can have $co = co'$ as a valid justification order for E.

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 $a_x \xrightarrow{hb} a_y$. Thus, we need to show that a_y can occur presciently after α.

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.

If a_y is a write, then all read actions that conflict with it and happen before it must be in α. Since a_y does not induce any interthread happens-before orderings, the only way for an action b to happen before a_y is if b happens-before an action that occurs earlier than a_y in program order.

- The action b cannot be a_x, because a_x and a_y do not conflict.
- The action b can also not be in β. Since a_x is a write, anything that happens before it must also happen before a_y. Therefore, a reversal of a_x and a_y would have removed the happens-before edge between b and a_x.

The action b must, therefore, still be in α.

- If neither of those cases hold, then $co = \alpha a_x \beta a_y \gamma$ (and it doesn't matter if a_y is a read or not). We will show that we can have $co = co'$ as a valid justification order for E. 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 reperformed 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 *co*. Since *co* is non-prescient, the ordering of the actions in *co* 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 conflict 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, $\alpha = \emptyset$. All non-prescient extensions of $\emptyset$ 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 extensions 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.

    There must be a non-forbidden $E' \in NPE(\alpha\beta')$ that includes an x' that corresponds to x; otherwise, it would not be possible to include it as the next action in the justification order.

    The action x does not influence whether r occurs or the variable it reads; it only influences the value it sees. All non-prescient extensions of $\alpha\beta'$ must therefore contain a read corresponding to r, reading the same variable, but possibly seeing a different value. Let r' be this corresponding read in E', and let E'' be the full prescient relaxation of E'.

    Because β' 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 both E'' and E'.

Because E' is correctly synchronized, r' and x' must be ordered by happens-before edges in E':

* $x' \stackrel{hb'}{\rightarrow} r'$: Since r' is in all non-prescient extensions of $\alpha\beta'$, and can occur directly after $\alpha\beta'$, the only actions that can have happen before edges leading to r' are in $\alpha\beta'$. Since x' is not in $\alpha\beta'$, we cannot have $x' \stackrel{hb'}{\rightarrow} r'$.
* $r' \stackrel{hb'}{\rightarrow} x'$: The presence of r' mandates that $r \stackrel{hb}{\rightarrow} x$ in E. But since r sees x, this is impossible.

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 are sequentially consistent. □

Out-Of-Thin-Air Reads are Illegal

We have three criteria for determining whether a reference to an object is out-of-thin-air:

- The thread that allocated that object can write the reference before reading it.
- Any thread that did not allocate the object, but writes a reference to it, must read that reference before writing it.
- If an address is written by any thread, there must be a write of that address by the allocating thread.

Assume that the first write action that violates these rules is a write action w; it writes an *out-of-thin-air* reference. The action w can only occur in a justification order $\alpha w \beta$ if it occurs in every non-prescient extension of α. This implies that w is not allowed to write any other reference in any other justifying execution.

If w is a write by the thread that allocated the object, it is not out of thin air. Therefore, w must occur in another thread, and must be preceded in program order by a read r that produces this reference. This allows for one of two possibilities:

- There is no non-prescient extension of α in which any other reference is written. The reference written is completely determined by α. In this case, the reference written was produced by some read in α; this follows our second criterion.
- All non-prescient extensions of α in which the reference written is not the one being written out here are forbidden. For this to happen, the read r on which this write is dependent must, in all non-prescient extensions of α, return the out of thin air reference. This implies that all executions in which r returned a different write must be forbidden.

The action r can occur immediately after α, because w is dependent on it, and w is allowed to occur immediately after α. There is therefore a forbidden execution with $\alpha r' w'$, where $\alpha r'$ is the forbidden prefix and r' does not return the out-of-thin-air reference. There must therefore be a non-forbidden prefix $\alpha r''$ where r'' returns another value determined by α. The write w' must be allowed to write this value in that non-forbidden execution; therefore, it is not the case that the write must return only the out-of-thin-air value in every non-prescient extension of α. This is a contradiction.