First, we will look at an example from previous discussions with SG1 committee members when looking at memory models. In the execution above, the absence of dependency makes it possible for Thread 1 to load before Thread 0 has written. The standard must allow the optimisation. Syntactic dependency, the curved arrows in the executions are make up the nodes, the black edges represent dependency edges from the loads to the stores in the execution with outcome where the compiler must be free to optimise, so there can be no restriction imposed on the optimiser. Hotspot will optimise Thread 1 as follows: the load of x will be done before the store of y, because concrete values were written making the writes independent. By forbidding cycles in dependency and control dependency edges over the memory model, called the Goldblatt alignment, the compiler can remove the conditions in the outcome where.

As an implementation-defined relation, restricted by legal values in x and y, the Goldblatt alignment permits this behavior even on targets whose architectures forbid the relaxed behavior. Target architectures like ARM and POWER allow Load buffering with false-dependency and the execution should be forbidden. We do not have a sequence of optimisations that leads to this behavior, we believe this execution should be forbidden, and the outcome where.

The curved arrows in the executions are make up the nodes, the black edges represent dependency edges from the loads to the stores in the execution with outcome where the compiler must be free to optimise, so there can be no restriction imposed on the optimiser. Hotspot will optimise Thread 1 as follows: the load of x will be done before the store of y, because concrete values were written making the writes independent. By forbidding cycles in dependency and control dependency edges over the memory model, called the Goldblatt alignment, the compiler can remove the conditions in the outcome where.

We do not have a sequence of optimisations that leads to this behavior, we believe this execution should be forbidden. With this constraint, the compiler can remove the conditions in the outcome where.

The standard must allow the optimisation. Syntactic dependency, the curved arrows in the executions are make up the nodes, the black edges represent dependency edges from the loads to the stores in the execution with outcome where the compiler must be free to optimise, so there can be no restriction imposed on the optimiser. Hotspot will optimise Thread 1 as follows: the load of x will be done before the store of y, because concrete values were written making the writes independent. By forbidding cycles in dependency and control dependency edges over the memory model, called the Goldblatt alignment, the compiler can remove the conditions in the outcome where.

We do not have a sequence of optimisations that leads to this behavior, we believe this execution should be forbidden.

The curved arrows in the executions are make up the nodes, the black edges represent dependency edges from the loads to the stores in the execution with outcome where the compiler must be free to optimise, so there can be no restriction imposed on the optimiser. Hotspot will optimise Thread 1 as follows: the load of x will be done before the store of y, because concrete values were written making the writes independent. By forbidding cycles in dependency and control dependency edges over the memory model, called the Goldblatt alignment, the compiler can remove the conditions in the outcome where.

We do not have a sequence of optimisations that leads to this behavior, we believe this execution should be forbidden.

The curved arrows in the executions are make up the nodes, the black edges represent dependency edges from the loads to the stores in the execution with outcome where the compiler must be free to optimise, so there can be no restriction imposed on the optimiser. Hotspot will optimise Thread 1 as follows: the load of x will be done before the store of y, because concrete values were written making the writes independent. By forbidding cycles in dependency and control dependency edges over the memory model, called the Goldblatt alignment, the compiler can remove the conditions in the outcome where.

We do not have a sequence of optimisations that leads to this behavior, we believe this execution should be forbidden.

The curved arrows in the executions are make up the nodes, the black edges represent dependency edges from the loads to the stores in the execution with outcome where the compiler must be free to optimise, so there can be no restriction imposed on the optimiser. Hotspot will optimise Thread 1 as follows: the load of x will be done before the store of y, because concrete values were written making the writes independent. By forbidding cycles in dependency and control dependency edges over the memory model, called the Goldblatt alignment, the compiler can remove the conditions in the outcome where.

We do not have a sequence of optimisations that leads to this behavior, we believe this execution should be forbidden.
4.9. 

Forbidden, cycle in

4.8. 

Allowed, no cycle.

4.7. 

Forbidden, cycle in

4.6. 

Allowed, no cycle. Similar to above.

4.5. 

Allowed, no cycle. A compiler can observe that

Is SG1 happy with this set of transformations? Should this weak execution be allowed?

conditional. The LLVM trace preserving transformation

value that may lead to undefined behaviour.

We

Load forwarding

We introduce a non-atomic load of

The optimisations are all on Thread 1, and we repeat the transformed program after each step. To start with,

make assumptions about the values read by non-atomics. Use of
4.20. We skip this as it probes thread joining, which is out-of-scope for this paper.

4.19. We skip this as it involves loops.

4.18. We skip this as the test probes Java's weak coherence-order guarantees, and is not relevant to C++.

4.17. We skip this as the test probes Java's weak coherence-order guarantees, and is not relevant to C++.

4.16. We skip this as the test probes Java's weak coherence-order guarantees, and is not relevant to C++.

4.15. We skip this as it involves loops.

4.14. Allowed. Similar to test case 8, except that the x is not always 0 or 1. However, a compiler might determine (that the read of x by thread 2 will never see the write by thread 3 (perhaps because thread 3 will be scheduled after thread 1).

4.13. Allowed, no cycle. Observable through compiler and hardware re-ordering.

4.12. Forbidden, cycle in

4.11. Allowed, no cycle.

4.10. Forbidden, cycle in

4.09. Allowed, no cycle.