LoopPredication

Loop Predication

Loop Predication is an optimization in C2. The general idea is to insert a predicate on the entry path to a loop, and raise a uncommon trap if the check of the condition fails. The condition checks are promoted from inside the loop body, and thus the checks inside the loop could be eliminated. Currently, loop predication optimization has been applied to remove array range check and loop invariant checks (such as null checks and array checks).

Array Range Check Elimination

Use loop predicate to remove array range checks in a loop.

original loop

```
for (int i = init,  i < limit;  i += stride) {  // loop with array range check
    ...
    if (scale*i + offset < a.length) {  // array range check
        ... a[scale*i+offset] ...
    } else raise_uncommon_trap();
    ...
}
```

after loop predication

```
if ( scale * imax + offset < a.length ) {  // loop predicate. imax is the maximum value of i where init <= i < limit
    for (int i = init,  i < limit;  i += stride) {  // loop without array range check
        ...
        ... a[scale*i+offset] ...
        ...
    }
} else raise_uncommon_trap();
```

From the above example, the requirements to perform loop predication for array range check elimination are that

init, limit, offset and array a are loop invariants, and stride and scale are compile time constants.

Loop Invariant Check Elimination

Similar to array range check elimination, but the loop invariant check elimination is much simple. The check inside the loop is loop invariant, and can be promoted outside the loop directly. Current implementation of loop invariant check elimination includes null check elimination and array check elimination, etc.

Loop Predication Advantages

Existing optimizations that perform elimination of checks inside the loops are *iteration range splitting* based. The loop is peeled to form pre-, main- and post-loops. The loop bounds are adjusted in the ways that checks in the main-loop could be safely removed.

The major disadvantage of *iteration range splitting* based check elimination is the dramatic increase in code size due to the copies of the loop. In addition, the checks in the pre- and post-loops are not eliminated, and thus the performance gain is limited when the iteration space is small.

Compared with the *iteration range splitting* based check elimination, loop predication has the following advantages:

1. Loop predication can be applied to outer loops without code size increment.

2. With loop predication, the check can be eliminated in the entire iteration space.

3. Loop predication can be applied to loops with calls

4. (for future work) Loop predication provides a unified interface that can be implemented (extended) to replace the existing *iteration range splitting* based check eliminations (loop peeling, partial peeling, range check elimination)

Source Code

1. The parser inserts a dummy loop predicate on each path to the loop head block (Parse::add_predicate in parse1.cpp)

2. The loop optimizer analyzes the loop and promote the checks outside the loop (IdealLoopTree::loop_predication in loopTransformation.cpp)

3. Loop predication is controlled by the -XX flag UseLoopPredicate. -XX:+TraceLoopPredicate turns on the debug engine for loop predication in debug VM.