Loop optimizations in Hotspot Server VM Compiler (C2)

Loop Optimizations in C2

Loop Identification

ciTypeFlow analysis

Single entry loops

Irreducible loops

Clone loop head

Parse phase

Add regions and phis for loop head

IdealLoop phase

Replace Region with Loop node

Beautify loops: split shared headers and merge multiple entry and back branch paths

Loop Construction

Building side tables for ideal graph

IdealLoopTree structures describe loops tree

Dominators for control graph

Data nodes placement

Initial nodes assignment to loops (build_loop_early)

Find counted loops

- Single entry loop and single back branch
- Integer iterative variable, constant stride, loop invariant limit
- replace Loop node with CountedLoop node and loop's condition If node with CountedLoopEnd node

Find best nodes placement (build_loop_late)

Loop Optimizations

Split-if and split-through-phi

Local subgraph transformations to simplify graph

Loop predication

Range checks and NULL checks moved outside loop if possible

Array filling

Pattern match array initialization loop and replace it with call to assembler stub

Vectorization

Replace array initialization, copy and arithmetic with vector operations in unrolled loops

Iteration splitting for inner loops

For each optimization below policy method is called to determine which optimization could be done for this loop

compute exact loop trip count if possible (constant limits)

convert one iteration loop into normal code

remove empty loops:

int i = 0; for(i<limit;i++){} converted to i=limit;

partial peeling (loop rotation) for non-counted loops

peeling and unswitching for non-counted loops

unswitching for counted loops (in some cases needs peeling also)

completely unroll counted loop with small number of iterations

- always unroll with 3 and less iterations
- limited by body size (LoopUnrollLimit flag)

iteration splitting for counted loops

- clone loop body and create pre-, main- and post- loops

Range Check Elimination

- Adjust limits on pre and main loop to avoid range check in main loop

Main loop unrolling (with maximum 16 unrolls)

Range Check Elimination

RangeCheckElimination

For array accesses range check is generated in compiled code:

```
If (index < Array.length) { // unsigned compare
    Array[index] = 0;
else
    uncommon_trap(range_check);
```

For loop-invariant arrays, range checks can usually be eliminated.

This is carried out by means of iteration range splitting. The middle (main-) loop handles the middle range, and the pre-loop (resp. post-loop) handles any index values before (resp. after) the middle range. The middle range is chosen so that the middle loop (main loop) is as large as possible, but is constrained to values which can be predicted not to cause array range checks to fail.

Here is a simple case:

```
for (int index = Start; index < Limit; index++) {
  Array[index] = 0;
}
```

The loop is splitted like this:

```
int MidStart = Math.max(Start+1, 0);
int MidLimit = Math.min(Limit, Array.length);
int index = Start;
for (; index < MidStart; index++) { // PRE-LOOP (executed at least once)
  Array[index] = 0; // RANGE CHECK generated
}
for (; index < MidLimit; index++) { // MAIN LOOP
  Array[index] = 0; // NO RANGE CHECK
}
for (; index < Limit; index++) { // POST-LOOP
  Array[index] = 0; // RANGE CHECK generated
}
```

More complex example. Array and Offset are loop invariants and scale in index expression is constant:

```
for (int index = Start; index < Limit; index += STRIDE) {
  ... Array[index * SCALE + Offset] ...
```

```
}
```

The values MinStart and MinLimit are computed similarly as above:

```
int MidStart = Math.max(Start+1, 1 - Offset/SCALE);

int MidLimit = Math.min(Limit, (Array.length-Offset)/SCALE);
```

It works with several arrays in loop by calculating new limits based on limits from previous RCE expressions.

This optimization also applies for condition expressions in a loop:

```
for (int index = Start; index < Limit; index += STRIDE) {

  If (index * SCALE + Offset < Range) {
```

Loop Predication

LoopPredication

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).

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

```
If ((Limit-1) * SCALE + Offset >= Array.length ||

    Start * SCALE + Offset < 0)

    uncommon_trap(predicate);

for (int index = Start; index < Limit; index += STRIDE) {

... Array[index * SCALE + Offset] ...

 }
```

Similar to array range check elimination loop predication is used to move a loop invariant check outside the loop directly.

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, loop predication has the following advantages:

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

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

3. Loop predication can be applied to loops with calls