RangeCheckElimination

Iteration range splitting

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

This is carried out by means of iteration range splitting. A middle range of loop index values is computed before the loop is entered. (It is often the whole sequence of index values, but need not be if loop peeling or unrolling is happening also.) The loop is cloned three times, with the three clones running in succession. The middle 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:

Simple Loop

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

The loop is split like this:

Simple Loop, Optimized

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

When the optimization applies

This typically happens when:

- the array is a loop invariant,
- on a hot loop,
- whose index variable has a constant stride,
- and where the array is indexed by linear functions of the index variable.

Typical Loop

```
for (int index = Start; index < Limit; index += STRIDE) {
  ... Array1[index * SCALE1 + Offset1] ...
  ... Array2[index * SCALE2 + Offset2] ...
}
```

Capitalized names (`Limit`, `Array1`) must be loop-invariant, while all-caps names (`STRIDE`, `SCALE1`) must be compile-time constants. Any number of arrays can be dealt with this way, both for read and write. Other non-matching arrays will not disturb the optimization, but they will be range-checked continually. The loop need not be a literal Java `for` statement; the JIT can deal with a wide range of loop idioms.

The values `MinStart` and `MinLimit` are computed similarly as above, except that there are more min/max terms, and there may be divisions if `SCALE` values are not unity.

This pattern is quite general, although it doesn't help with every imaginable loop. Pointer-chasing loops, and loops with non-constant strides,

don't participate in this optimization. Key rules of thumb:

- Make your array be loop invariant, typically by loading it into a local variable.
- Use at most simple linear expressions to index each array.

A constant value can be a `final static` variable.

A loop-invariant value can be an object field, as long as the compiler can prove the field isn't changed by the loop. This will fail if the loop body is not totally inlined, and the object field cannot be proven non-escaping.

Source code

- See logic related to `PhaseIdealLoop::add_constraint` in loopTransform.cpp.
- The optimization is controlled by the flag RangeCheckElimination.