Non Temporal Moves

# Owner and Point of Contact.

Mark Mendell (mark.p.mendell@intel.com)

# Idea of optimization: What is it? If a loop only is doing stores, it is possible to use the non temporal store instruction *movnti* to perform the stores and sink a memory barrier instruction at the end of the loop.

# Idea of optimization: Why do we need it?

Loops are often used to initialize arrays and the loops are very simple and straightforward. However, using normal mov instructions is not optimal in such cases, as the entire cache will become full with the array stores, tossing out other useful data. Instead, using the non-temporal version provides better performance.

# Idea of the optimization: How is it implemented?

The algorithm first starts by checking if the loop passes the conditions explained in the next section. If the loop does pass the checks, the optimization finds the array store operations and checks if the array stores are using the induction variable as an index.

For those, it will request the stores become non-temporal.

# Conditions for the optimization to work

The pass only applies to certain parts of loops:

* There must be no array loads in the loop
* The loop must be the inner loop
* The loop cannot throw an exception
* The increment of the loop must not be different than 1
* The loop must have constant bounds
* The loop must have a known number of iterations
* The loop must not have a floating point induction variable
* We only do it for loops with at least 131072 iterations
  + This bound was calculated as an estimation for having cache overfill and thus the moves can be streaming.
  + When testing, the number of iterations this optimization applies will have a margin of error of a couple of iterations because iteration peeling (pre and post loop) may be done.
* The store accesses need to use the induction variable as the index

There also needs to be array stores for the optimization to be applied.

# Limitation of optimization applicability

* The optimization only applies itself to array stores.
* Only applies to long, object, or int type of array accesses

# Examples in Java

public static int foo(int[] tab, int n) {

int m = 0x40000;

for (int i = 0; i < m; i++) {

tab[i] = i;

}

return n;

}

# Specific test cases that should be covered if known

None.

# Recommendation on testing focus

Test variations of the example. Use different types than int, object, or long, it should not apply in those cases.

Test with multiple array stores and check it happens to each.

# Post processing recommendations

If the optimization does apply itself, it should print out via the *--print-passes=non\_temporal\_move* option:

# … non\_temporal\_move: Mark NonTemporal ArraySet (TYPE) *v54 ArraySet [ l129 i113 j43 ]*

Where **TYPE** is one of int/long/object and the text after the ‘)’ will vary.

Note that the only way to detect the actually generated *movnti/movntiq* instructions is to use *oatdump* on the generated OAT file, and grepping for *movnti/movntiq.* For long stores, in 32 bit mode, there will be 2 *movnti* instructions per array store. In 64 bit mode, one *movntiq* instruction will be used. Object references are 32 bits long, and will always use one *movnti* instruction.

# Whether we need negative test cases

No

# Future Work

- Further increase the scope of the work to allow more array accesses to be used.