Constant Calculation Sinking

# Owner and Point of Contact.

Jean Christophe Beyler

# Idea of optimization: What is it?

Constant calculation sinking tries to detect calculations in a loop that are constant based. For example:

for (i = 0; i < 500; i++){

a += 5;

b += i;

}

Because the number of iterations are known, the variable *a* is not being used by anything else and the element adding is a constant, we can transform the loop into:

for (i = 0; i < 500; i++) {

b += i;

}

a += 2500;

Constant calculation sinking’s optimization is about performing optimizations such as the one above.

# Idea of optimization: Why do we need it?

There are cases where the calculation can be done at compile time instead of generating code, which we already know the answer to.

# Idea of the optimization: How is it implemented?

The implementation finds any variable that is calculated via an accumulation style system. This means any update to the variable such as:

a += cst;

b \*= cst;

The current implementation supports addition, subtraction, remainder, division, and multiplication. For the variable to be sunk and pre-calculated, it must not be used again by the loop and the element to the right of the ‘=’ must be a constant.

The types accepted are integer, float based types.

Note:

1. We expect no sinking occurs if int/long multiplication overflows.
2. For Float/double, we expect sinking occurs on following conditions:
   1. One dimensional loop:
      1. For addition/subtraction: if the number of iterations is less than 65 or overflows.
      2. For division/multiplication: if the number of iterations is less than 65 or overflows or calculation result is 0
      3. For reminder, always sink.
   2. Nested loop:
      1. addition/subtraction: if result overflows
      2. multiplication: if (overflows or result is 0) and (constant operand cst >= 0)
      3. division : if (overflows or result is 0) and (constant operand cst > 0 )
      4. reminder, always sink.
3. all sinking occurs only when target loop is not optimized away by Trivial Loop Evaluator optimization.

# Conditions for the optimization to work

The variable must be a local variable and not be used again in the loop, they must be of the type *variable operation= constant*. The types accepted are integer based types.

There cannot be two instructions updating twice the variable:

A \*= 5;

A += 4;

The bounds must be known at compile time.

# Limitation of optimization applicability

We do not handle byte/short, we do not handle the cases where it is not a constant but an invariant, we do not handle the case where the loop iteration count is not known at compile time even if it is an invariant.

The loop should only have two BasicBlocks, one body BB and one loop header BB, or one block for bottom-tested loop.

The loop cannot throw exceptions.

The loop should have known iteration numbers.

The loop should not in debuggable mode.

All these issues could be solved with a more generic version.

# Examples in Java

* Cases like the loop in CFBench MIPS

public class Example

{

public static void main (String[] args)

{

// We should test with a starting at a different value then 0

// And when a is in an if body in the loop

int a = 0;

for (int i = 0; i < 5000; i++)

{

a += 5; // or %=, -=, \*=, /=

}

}

}

# Specific test cases that should be covered if known

We should test if:

* The iteration count is not known at compile time
* The right-side of the instruction is not a constant such as a parameter (a variable set to a constant and using that constant should be supported)
* The instruction must be executed per iteration (so cannot be in an if)
* The variable used multiple times

# Recommendation on testing focus

Testing variations of the instructions, in ifs, types, etc.

# Post processing recommendations

The optimization prints out message if the innermost loop was rejected by CCS:

Constant Calculation Sinking: Loop with head bb x is not good for CCS. where x is the loop header block number.

And one message if it sinks an instruction:

"Successfully sunk %s"

# Whether we need negative test cases

No

# Future Work

Work on updating with invariants instead of only constants.

Work on instructions when iteration count is fixed but not defined by a constant.