Constant Calculation Sinking Test Design Document

# 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 int, and long.

# 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 type of the variable cannot be byte/short.

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 one BasicBlock.

The loop cannot work if there are suspend checks.

The loop cannot throw exceptions.

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 various messages:

Constant Calculation Sinking: Not applicable because X:

* Reasons being:
  + Must be in the inner loop
  + Has no suspend check
  + Must not throw an exception
  + Only one BB
  + Unknown number of iteration

And one message if it sinks an instruction:

"Successfully sunk %s"

# Post processing implementation

Post-processing script checks that logcat contains expected number of "Constant Calculation Sinking: Successfully sunk" messages for each test method.

# Test cases covered

* Positive tests for int/long types
* Different kinds of variable to be sunk: local variable, instance/static field, volatile, array element, …
* Several loops to be optimized in a row
* Several variables to be sunk in one loop
* Large loop iteration count, small loop iteration count
* for/while/do-while and combinations
* Count down loop, IV other than 1, float-point IV, multiple IVs
* Nested loops: optimization should be applied to inner loop only
* Combination with other loop optimizations, e.g., non temporal move (can’t be applied together??), AccumulationSinking
* Negative tests for byte, short, float, double, char
* All supported operations: addition, subtraction, remainder, division, and multiplication
* Negative tests: variable used again in the loop
  + There cannot be two instructions updating twice the variable: A \*= 5; A += 4;
* Negative tests with bounds unknown at compile time (but invariant)
* Positive tests for known bounds starting with a number other than zero
* Negative tests for non-constant invariant in the right side (such as parameter)
* Positive test for variable set to a constant in the right side
* Negative tests breaking one basic block limitation: function invocations, exceptions, if-else, switch
* Test with casts within the loop
* Check the operations that are not supported (e.g., >>); case v = x – v , case v = x/v.
* Enhanced for loop (not supported)
* Break/continue

# To be covered in future

* Multi-thread test case
* Negative tests breaking no suspend check condition
* What if we overflow when multiplying? Calculation must be correct
* More complex cases of variable to be sunk usage before and after the loop