**15-745: Optimizing Compilers**

**Assignment 1 Writeup**

**Ben Humberston, bhumbers**

**Prashanth Suresh, psuresh**

**Implementation Summary**

**FunctionInfo:**

All implementation is found in FunctionInfo::printFunctionInfo(). We iterate through all functions in the module and calculate statistics for each independently. This includes an inner iteration for call counts where we examine all instructions in the module to test whether it matches the current function under consideration; this could be optimized by collecting the call counts for all functions at once in a single pass through the module, but we consider the unoptimized form workable.

**LocalOpts:**

All implementation is found in LocalOpts::applyLocalOptimizations(). Until we find that no more optimizations are possible we repeatedly iterate through all instructions in the module and apply algebraic identity simplifications, constant folding, and power reduction optimizations. We handle cases where only one operand is a constant in either the first or second operand position using condition-dependent pointers to the constant and non-constant terms

**Test Listing**

(Note: Only listing tests additional tests implemented by our group)

* ./FunctionInfo/loop.c
  + Fact() & call(): Tests behavior on a slightly more complex, recursive function
* ./FunctionInfo/tests/var\_args\_test.c
  + var\_args\_func(): Verifies correct identification of variable argument function
* ./LocalOpts/test-inputs/algebraic.c
  + add\_zero(), mul\_one(), div\_one(): Verify simple algebraic identity behavior
  + alg\_identity\_combo(): Verifies optimization behavior for slightly more complex algebra
  + alg\_const\_folds(): Verifies optimization behavior for multi-pass constant folding
  + undef\_var\_test(): Verifies that crash doesn’t occur when one term is undefined

**Test Output**

See README.txt for shell commands to run all tests

**FunctionInfo Expected Test Outputs**

15745 Function Information Pass

Module loop.bc

Name, Args, Calls, Blocks, Insns

g\_incr, 1, 0, 1, 4

loop, 3, 0, 3, 10

Fact, 1, 0, 6, 41

var\_ar, \*, 0, 1, 2

call, 0, 0, 1, 1

**LocalOpts Expected Test Outputs**

Module loop-m2r.bc

Optimized an instance of additive identity.

Transformations Applied:

Algebraic Identities: 1

Constant Folding: 0

Strength Reduction: 0

Module algebraic-m2r.bc

Optimized an instance of additive identity.

Optimized an instance of multiplicative identity.

Optimized an instance of division identity.

Optimized an instance of division identity.

Optimized an instance of multiplicative identity.

Const-folded an expression: mul 6, 7

Const-folded an expression: add1 6, 42

Optimized an instance of multiplicative identity.

Optimized an instance of additive identity.

Const-folded an expression: sub 42, 1

Const-folded an expression: add 41, 3

Transformations Applied:

Algebraic Identities: 7

Constant Folding: 4

Strength Reduction: 0

Module constfold-m2r.bc

Const-folded an expression: add 4, 2

Const-folded an expression: add2 0, 2

Const-folded an expression: add1 6, 3

Const-folded an expression: add3 2, 3

Const-folded an expression: mul 5, 9

Transformations Applied:

Algebraic Identities: 0

Constant Folding: 5

Strength Reduction: 0

Module strength-m2r.bc

About to apply a bitshift: 2, 1

Optimized an instance of additive identity.

About to apply a bitshift: 8, 3

Transformations Applied:

Algebraic Identities: 1

Constant Folding: 0

Strength Reduction: 2

**5.1 CFG Basics**

Basic blocks:

**B1:** x = 100

y = 0

goto L2

**B2:**  L1: y = x \* y

if (x < 50) goto L2

**B3:** y = x - y

goto L3

**B4:** L2: y = x + y

**B5:** L3: print(y)

if (y < 1000) goto L1

**B6:** switch (x) { 0 => L6 |

**B7:** 1 => L4 |

**B8:** 101 => L7

**B9:**  default => L5 }

**B10:** L4: print("!")

**B11:** L5: x = x - 1

goto L1

**B12:** L6: return y

**B13:** L7: goto L7

**B1:** x = 100

y = 0

goto L2

**B5:**

L3: print(y)

if (y < 1000) goto L1

**B6:**

switch (x) { 0 => L6 |

**B7:**

1 => L4 |

**B8:**

101 => L7

**B3:**

y = x - y

goto L3

**B2:**L1: y = x \* y

if (x < 50) goto L2

**B9:**

default => L5 }

**B12:**

L6: return y

**B13:**

L7: goto L7

**B10:**

L4: print("!")

**B11:**

L5: x = x - 1

goto L1

**B4:**

L2: y = x + y

**5.2 Available Expressions**

|  |  |  |
| --- | --- | --- |
| **BB** | **EVAL** | **KILL** |
| 1 | b+c, b\*b, b\*d |  |
| 2 | b+c, i+1 |  |
| 3 | b\*b |  |
| 4 | b\*b | b\*d |
| 5 | i+1 |  |

|  |  |  |
| --- | --- | --- |
| **BB** | **IN** | **OUT** |
| 1 | {} | b+c, b\*b, b\*d |
| 2 | b+c, b\*b, b\*d | b+c, b\*b, b\*d, i+1 |
| 3 | b+c, b\*b, b\*d, i+1 | b+c, b\*b, b\*d, i+1 |
| 4 | b+c, b\*b, b\*d, i+1 | b+c, b\*b, i+1 |
| 5 | b+c, b\*b, i+1 | b+c, b\*b, i+1 |

**5.3 Faint Analysis**

Note: The following data flow pass is almost exactly the same as the liveness pass from Lecture 4, with the exception that the “Use” set in the transfer function does not contain locally exposed uses where the variable being used is itself on the LHS. Additionally, the way we mark faint/dead vs. live variables at each code point (part 9) is modified to ensure that faint variables are correctly marked within a block.

We considered an alternative algorithm which had an additional constraint on insertion into the “Use” set stating that the variable on the LHS of each use must be in the live set being provided in the block’s edge (backward analysis). If were a non-empty set (eg, if it consisted of the variables to be returned from a function), we believe this algorithm would correctly mark as faint all variables which are not used to calculate the members of that set. However, because , adding this constraint would have the effect of marking *all* variables in all blocks as faint (since all variables are dead/faint at exit), so all assignments would be to faint variables and may be eliminated!

Thus, we do not include this constraint and allow faint variables within a block to be considered “live” in predecessor blocks, so that the faintness condition only applies within blocks, not between them. This seems to match the intent of the two cases provided in the problem description.

**1.** Set of elements: Set of live *variables*

**2.** Direction:Backward analysis

**3.** Transfer Function:

Here, is the set of variables defined in basic block and is the set of locally exposed uses in , except where the usage has the variable itself in the LHS of the assignment (this prevents adding a variable to the live set when its only future LHS “user” is itself).

**4.** Meet Operator: Union .

(cont’d on next page)

**5.** Boundary Condition:

**6.** Initial interior points: for all interior basic blocks

**7.**  Block Ordering:

It is expected that blocks will be visited from back to in a reverse breadth-first manner. Since liveness of a variable depends on the future blocks in a program, this back-to-front ordering ensures that the live set is complete for each block when it is considered.

**8.** Convergence:

This pass is guaranteed to converge. We repeatedly iterate over all blocks, applying the transfer function, until no changes occur to for any block . The maximum possible live set at each will be the universe of variables for the CFG. Since the transfer function which updates only removes elements if they are defined in a block, will grow monotonically on each repetition of the pass. Thus, convergence occurs when all sets reach their maximum (finite) size.

**9.** Faintness Algorithm:

For each variable in block , if the variable is neither a member of nor part of the RHS of an assignment to a variable that is in , then that variable/assignment is faint in that block. That is, if the variable is not part of nor does it affect some variable in the future live set specified at the edge of , then it is considered faint, as the assignment will have no impact on future code.

*For each basic block B {*

*For each variable V assigned in B {*

*If ( and for all variable assignments … which use V) {*

*Mark as faint*

*}*

*}*

*}*