Lecture 21: Code optimization tips

David Hovemeyer

November 16, 2022

601.428/628 Compilers and Interpreters

Today

- Implementing local value numbering
- Copy propagation
- Implementing register allocation

Implementing local value numbering

Local value numbering

Relatively easy way to find redundant computations

A *value number* is an integer representing a runtime value; if two locations are known to have the same value number, then at runtime they are guaranteed to have the same value

Optimization: replace recomputations with use of available values

Fundamental data structure: map of "LVN keys" to value numbers

LVN key

A LVN key identifies a computed value:

- Value numbers of operand(s)
  - For commutative operations, canonicalize the order (e.g., left hand operation must have lower value number)
- Operation being performed (add, subtract, etc.)
- Whether the computed value is a compile-time constant

Data to keep track of

As the analysis progresses, keep track of:

- map of constant values to their value numbers (just for constant values)
- map of value numbers to constant values (just for constant values)
- map of virtual registers to value numbers (i.e., find out what value number is in each virtual register)
- map of value numbers to sets of virtual registers known to contain the value number
- map of LVNKey to value number
- next value number to be assigned

Modeling instructions

- see an unknown vreg: assign a new value number
- see a load from memory, or a read: assign a new value number
- computed value: find value number of value being computed
  1. find value numbers of operands
  2. create an LVNKey from opcode and operand value numbers
     - canonicalize order of operands if operation is commutative
     - determine if value is a compile-time constant
  3. check map of LVNKey to value number; if not found, assign new value number (and update the map)

For each def (assignment to vreg), goal is to know the value number of value being assigned to the vreg

Effect of defs

If a def assigns a value number to a vreg that is different than the one it previously contained, then we must update all data structures appropriately.

Including: removing it from the set of vregs known to contain its previous value number.

Important! The value in a vreg should only be overwritten if it is being used as storage for a local variable. Temporary vregs allocated in expression evaluation shouldn't be overwritten, meaning values computed in expression evaluation should always be available.

Transformation

To the extent possible, every def of the form

vreg $\leftarrow$ *some value*

is replaced with

vreg $\leftarrow$ *known value*

"known value" could be a compile-time constant (best case), or a vreg known to store the same value as *some value*

What LVN achieves

Value numbering doesn't eliminate any instructions: it just makes redundancies more explicit.

Subsequent copy propagation and elimination of stores to dead vregs passes will remove instructions that are no longer needed.

Copy propagation

Result of LVN, copy propagation

LVN will generate instructions of the form

$$vreg_n \leftarrow vreg_m$$

where vreg_m is a virtual register containing a previously computed value

Subsequent uses of vreg_n can be replaced with vreg_m. This transformation is *copy propagation*.

Copy propagation example

Consider the code:

```
/* original code */
add_l vr2, vr0, vr1
mov_l vr4, vr0
add_l vr5, vr4, vr3
```

Copy propagation example

After copy propagation:

```
/* original code */          /* transformed code */
add_l vr2, vr0, vr1          add_l vr2, vr0, vr1
mov_l vr4, vr0               mov_l vr4, vr0
add_l vr5, vr4, vr3          add_l vr5, vr0, vr3
```

If `vr4` became dead at the point of the assignment to it, the `mov_l` instruction can be eliminated

Implementing local register allocation

Local register allocation

Goal of register allocator is to assign (temporarily!) machine registers to virtual registers

- Special case: virtual registers that are alive at the end of the basic block shouldn't have a temporary register assignment
- Could leave these allocated in memory, or use "long term" register assignment (i.e., callee-save registers)

Problem: there are a limited number of machine registers

- If we run out, steal a machine register that is currently in use, first spilling its value to memory
- Bottom-up register allocation: when stealing, choose the virtual register whose next def is the furthest in the future

Register allocator state

Information to keep track of as allocator progresses through instructions in basic block:

- Collection of available machine registers (stack or queue)
- Map of virtual register numbers to assigned machine register
- Collection of available spill locations (stack or queue)
- Map of virtual register numbers to spill locations

Recording register assignments

The register allocator will need to communicate register assignments to the low-level code generator.

One way to do this: add a field to Operand, if set to a non-negative value, it's the assigned machine register.

Making allocations and assignments

For each virtual register used in an instruction[1]:

- If its value is currently spilled, allocate a machine register and restore it
- If there is a current assignment to a machine register, record the assignment
- If there is no assignment, allocate a register and record the assignment

[1] Except for vregs excluded from being assigned a temporary register.

Allocating a register

- If a machine register is available, allocate it (easy case)
- If no machine register is available (harder case):
  1. Choose a victim vreg
  2. Allocate a currently-unused spill location, otherwise, use a new spill location
  3. Emit a spill instruction (specifying the vreg, mreg, and spill location)
  4. Use the stolen mreg to satisfy the allocation

Restoring a spilled register

Assuming that an machine register has already been allocated:

1. Emit a restore instruction (specifying vreg, mreg, and spill location)
2. Return the (no longer used) spill location to the collection of available spill locations

Allocating storage for spill locations

Determine maximum number of spill locations used (over all basic blocks)

Place storage area for spills somewhere in the stack frame

Low-level code generator will need to determine an offset into the storage area for each spill and restore

Procedure calls

The compiler must assume that a call to a procedure could change the value of any caller-save register! (e.g., %rcx, %rdx, %r10, etc.)

Also, argument registers must be available for passing argument values to called function(s), so they might not be available for register allocation

Possible approach:

- A `call` instruction always ends a basic block (so there's at most one call per basic block)
- The register allocator should not allocate argument registers needed to pass values