Lecture 18: Implementing local value numbering, copy propagation

David Hovemeyer

November 27, 2023

601.428/628 Compilers and Interpreters

Today

- Implementing local value numbering
- Copy propagation

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