Lecture 18: Code optimization, local value numbering, copy propagation

David Hovemeyer

November 5, 2025

601.428/628 Compilers and Interpreters

Agenda

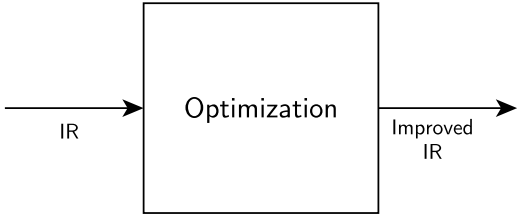
- Code optimization
- Local value numbering
- Implementing local value numbering
- Copy propagation

Code optimization

Code optimization

Code optimization refers to transformations to the IR intended to make the generated code "better"

- Usually, runtime execution speed is the goal
- Smaller code size and/or improved energy efficiency could also be goals



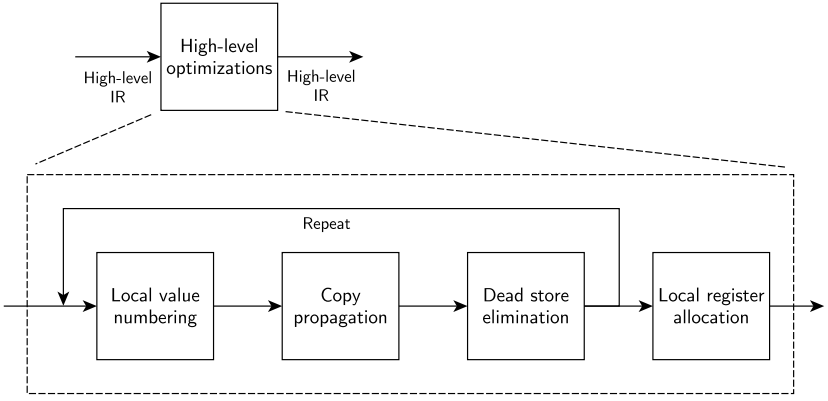
Performing optimization

- Optimization techniques: active area of research since the early days of compilers (1950s!)
- General considerations:
  - Which IR(s) to transform?
    - AST, high-level linear, low-level linear?
  - In what order should the transformations be applied?
  - How many times should transformations be applied?

Possible approach

Note that optimizations could be performed on the AST, but typically focus on the linear IR (both high- and low-level)

Ordering optimizations, iterating



Individual optimizations within a phase are executed in a sequence, and may benefit from being repeated.

How an optimization works, preserving correctness

An optimization will generally:

1. Perform an *analysis* to determine when, where, and how the optimization can be applied
2. Apply a *transformation* of the IR when the optimization has determined it can/should be applied

Optimizations must preserve the semantics of the source program as dictated by the language specification.

Analysis must be *conservative*: unless the optimization is 100% confident that a transformation preserves semantics, it can't apply the transformation.

Limits of optimization

- Optimizations generally don't result in "optimal" code
  - Although, modern optimization techniques generally produce code that is competitive with or superior to hand-written assembly language
- The goal is to *improve* the quality of the generated code
- This means that initial code generation generally does *not* need to be concerned about the efficiency of the generated code
  - Early code generation focuses on accurately representing the low-level operations that the program will carry out
  - It's often counterproductive to try to optimize early

Optimization scope

The *scope* of an optimization describes what regions of the program the optimization can analyze and transform.

Scope	Can analyze/transform	Example
Local	Single basic blocks	Local value numbering
Regional	Multiple BBs w/o control joins	Super LVN
Global	All basic blocks within a function	Liveness analysis
Interprocedural	All code in the program	Function inlining

Larger scopes require more powerful analysis techniques, but have more potential to improve the code. E.g., a local analysis might miss optimization opportunities that a global analysis could recognize and exploit.

Local value numbering

Local value numbering

Local value numbering (LVN) is a technique for identifying and eliminating redundant computations

Basic idea: within a basic block

1. Assign a *value number* to each value
2. If two locations (e.g., virtual registers) contain the same value number, then at runtime, they will contain the same value
3. If a computation is repeated (same operation applied to same values), it is *redundant* and can be replaced by a reuse of the previous computation of that value

LVN mechanics

"If a computation is repeated (same operation applied to same values), it is *redundant* and can be replaced by a reuse of the previous computation of that value."

What this means in practice is that when local value numbering notices that an available value is being recomputed, it replaces the computation with a move instruction that copies the original result into the destination of the recomputation instruction.

So, LVN doesn't eliminate any instructions: it just identifies redundant computations and *makes them explicit.* Subsequent "cleanup" passes will remove the unnecessary instructions.

LVN example

C code:

```
x = a * 3 + 4;
y = a * 3 - 5;
```

Assume: x is vr10, y is vr11, a is vr12, temporary vregs start at vr20

High-level IR:

```
mov_l   vr20, $3          /* constant 3 */
mul_l   vr21, vr12, vr20  /* a * 3 */
mov_l   vr22, $4          /* constant 4 */
add_l   vr23, vr21, vr22  /* a * 3 + 4 */
mov_l   vr10, vr23        /* assign to x */
mov_l   vr24, $3          /* constant 3 */
mul_l   vr25, vr12, vr24  /* a * 3 */
mov_l   vr26, $5          /* constant 5 */
sub_l   vr27, vr25, vr26  /* a * 3 - 5 */
mov_l   vr11, vr27        /* assign to y */
```

LVN example

```
mov_l   vr20, $3          /* constant 3 */
mul_l   vr21, vr12, vr20  /* a * 3 */
mov_l   vr22, $4          /* constant 4 */
add_l   vr23, vr21, vr22  /* a * 3 + 4 */
mov_l   vr10, vr23        /* assign to x */
mov_l   vr24, $3          /* constant 3 */
mul_l   vr25, vr12, vr24  /* a * 3 */
mov_l   vr26, $5          /* constant 5 */
sub_l   vr27, vr25, vr26  /* a * 3 - 5 */
mov_l   vr11, vr27        /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

Goal: assign a value number to each operand.

LVN example

```
mov_l  0vr20, 0$3          /* constant 3 */
mul_l    vr21, vr12, vr20  /* a * 3 */
mov_l    vr22, $4          /* constant 4 */
add_l    vr23, vr21, vr22  /* a * 3 + 4 */
mov_l    vr10, vr23        /* assign to x */
mov_l    vr24, $3          /* constant 3 */
mul_l    vr25, vr12, vr24  /* a * 3 */
mov_l    vr26, $5          /* constant 5 */
sub_l    vr27, vr25, vr26  /* a * 3 - 5 */
mov_l    vr11, vr27        /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

Constant value 3 is value number 0 (available in vr20).

LVN example

```
mov_l  0vr20, 0$3          /* constant 3 */
mul_l  2vr21, 1vr12, 0vr20 /* a * 3 */
mov_l   vr22, $4           /* constant 4 */
add_l   vr23, vr21, vr22   /* a * 3 + 4 */
mov_l   vr10, vr23         /* assign to x */
mov_l   vr24, $3           /* constant 3 */
mul_l   vr25, vr12, vr24   /* a * 3 */
mov_l   vr26, $5           /* constant 5 */
sub_l   vr27, vr25, vr26   /* a * 3 - 5 */
mov_l   vr11, vr27         /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

a is value number 1, a * 3 is value number 2 (available in vr21).

LVN example

```
mov_l  0vr20, 0$3          /* constant 3 */
mul_l  2vr21, 1vr12, 0vr20 /* a * 3 */
mov_l  3vr22, 3$4          /* constant 4 */
add_l    vr23, vr21, vr22  /* a * 3 + 4 */
mov_l    vr10, vr23        /* assign to x */
mov_l    vr24, $3          /* constant 3 */
mul_l    vr25, vr12, vr24  /* a * 3 */
mov_l    vr26, $5          /* constant 5 */
sub_l    vr27, vr25, vr26  /* a * 3 - 5 */
mov_l    vr11, vr27        /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

Constant value 4 is value number 3 (available in vr20).

LVN example

```
mov_l  0vr20, 0$3          /* constant 3 */
mul_l  2vr21, 1vr12, 0vr20 /* a * 3 */
mov_l  3vr22, 3$4          /* constant 4 */
add_l  4vr23, 2vr21, 3vr22 /* a * 3 + 4 */
mov_l   vr10, vr23         /* assign to x */
mov_l   vr24, $3           /* constant 3 */
mul_l   vr25, vr12, vr24   /* a * 3 */
mov_l   vr26, $5           /* constant 5 */
sub_l   vr27, vr25, vr26   /* a * 3 - 5 */
mov_l   vr11, vr27         /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

`a * 3 + 4` is value number 4 (available in `vr23`).

LVN example

```
mov_l  0vr20, 0$3          /* constant 3 */
mul_l  2vr21, 1vr12, 0vr20 /* a * 3 */
mov_l  3vr22, 3$4          /* constant 4 */
add_l  4vr23, 2vr21, 3vr22 /* a * 3 + 4 */
mov_l  4vr10, 4vr23        /* assign to x */
mov_l   vr24, $3           /* constant 3 */
mul_l   vr25, vr12, vr24   /* a * 3 */
mov_l   vr26, $5           /* constant 5 */
sub_l   vr27, vr25, vr26   /* a * 3 - 5 */
mov_l   vr11, vr27         /* assign to y */
```

a * 3 + 4 is assigned to x.

x is vr10
y is vr11
a is vr12
Temps start at vr20

LVN example

```
mov_l  0vr20, 0$3            /* constant 3 */
mul_l  2vr21, 1vr12, 0vr20   /* a * 3 */
mov_l  3vr22, 3$4            /* constant 4 */
add_l  4vr23, 2vr21, 3vr22   /* a * 3 + 4 */
mov_l  4vr10, 4vr23          /* assign to x */
mov_l  0vr24, 0$3            /* constant 3 */
mul_l    vr25, vr12, vr24    /* a * 3 */
mov_l    vr26, $5            /* constant 5 */
sub_l    vr27, vr25, vr26    /* a * 3 - 5 */
mov_l    vr11, vr27          /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

Constant value 3 is value number 0 (available already in vr20, now also available in vr24).

LVN example

```
mov_l  0vr20, 0$3          /* constant 3 */
mul_l  2vr21, 1vr12, 0vr20 /* a * 3 */
mov_l  3vr22, 3$4          /* constant 4 */
add_l  4vr23, 2vr21, 3vr22 /* a * 3 + 4 */
mov_l  4vr10, 4vr23        /* assign to x */
*mov_l 0vr24, 0vr20        /* constant 3 */
mul_l    vr25, vr12, vr24  /* a * 3 */
mov_l    vr26, $5          /* constant 5 */
sub_l    vr27, vr25, vr26  /* a * 3 - 5 */
mov_l    vr11, vr27        /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

Make the redundancy explicit by copying vr20 to vr24.

LVN example

```
mov_l  0vr20, 0$3          /* constant 3 */
mul_l  2vr21, 1vr12, 0vr20 /* a * 3 */
mov_l  3vr22, 3$4          /* constant 4 */
add_l  4vr23, 2vr21, 3vr22 /* a * 3 + 4 */
mov_l  4vr10, 4vr23        /* assign to x */
mov_l  0vr24, 0vr20        /* constant 3 */
mul_l  2vr25, 1vr12, 0vr24 /* a * 3 */
mov_l    vr26, $5          /* constant 5 */
sub_l    vr27, vr25, vr26  /* a * 3 - 5 */
mov_l    vr11, vr27        /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

Product of value number 1 (a) and value number 0 (constant 3) is already known to be value number 2 (available in vr21, now also in vr25).

LVN example

```
mov_l  0vr20, 0$3          /* constant 3 */
mul_l  2vr21, 1vr12, 0vr20 /* a * 3 */
mov_l  3vr22, 3$4          /* constant 4 */
add_l  4vr23, 2vr21, 3vr22 /* a * 3 + 4 */
mov_l  4vr10, 4vr23        /* assign to x */
mov_l  0vr24, 0vr20        /* constant 3 */
*mov_l 2vr25, 2vr21        /* a * 3 */
mov_l    vr26, $5          /* constant 5 */
sub_l    vr27, vr25, vr26  /* a * 3 - 5 */
mov_l    vr11, vr27        /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

Make the redundancy explicit by copying vr21 to vr25.

LVN example

```
mov_l  0vr20, 0$3          /* constant 3 */
mul_l  2vr21, 1vr12, 0vr20 /* a * 3 */
mov_l  3vr22, 3$4          /* constant 4 */
add_l  4vr23, 2vr21, 3vr22 /* a * 3 + 4 */
mov_l  4vr10, 4vr23        /* assign to x */
mov_l  0vr24, 0vr20        /* constant 3 */
mov_l  2vr25, 2vr21        /* a * 3 */
mov_l  5vr26, 5$5          /* constant 5 */
sub_l    vr27, vr25, vr26  /* a * 3 - 5 */
mov_l    vr11, vr27        /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

Constant value 5 is value number 5 (now available in vr26).

LVN example

```
mov_l  0vr20, 0$3          /* constant 3 */
mul_l  2vr21, 1vr12, 0vr20 /* a * 3 */
mov_l  3vr22, 3$4          /* constant 4 */
add_l  4vr23, 2vr21, 3vr22 /* a * 3 + 4 */
mov_l  4vr10, 4vr23        /* assign to x */
mov_l  0vr24, 0vr20        /* constant 3 */
mov_l  2vr25, 2vr21        /* a * 3 */
mov_l  5vr26, 5$5          /* constant 5 */
sub_l  6vr27, 2vr25, 5vr26 /* a * 3 - 5 */
mov_l   vr11, vr27         /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

a * 3 - 5 is value number 6 (now available in vr27).

LVN example

```
mov_l  0vr20, 0$3          /* constant 3 */
mul_l  2vr21, 1vr12, 0vr20 /* a * 3 */
mov_l  3vr22, 3$4          /* constant 4 */
add_l  4vr23, 2vr21, 3vr22 /* a * 3 + 4 */
mov_l  4vr10, 4vr23        /* assign to x */
mov_l  0vr24, 0vr20        /* constant 3 */
mov_l  2vr25, 2vr21        /* a * 3 */
mov_l  5vr26, 5$5          /* constant 5 */
sub_l  6vr27, 2vr25, 5vr26 /* a * 3 - 5 */
mov_l  6vr11, 6vr27        /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

Assign `a * 3 - 5` to `y`.

What did that accomplish?

In the two instructions where a previously-computed value was computed again, we changed the instruction to be a copy from the vreg holding the previously-computed value to the destination.

Next step is to perform *copy propagation*: within the basic block, if vreg A is copied to vreg B, replace uses of B with A.

If we eliminate all references to B, the instruction assigning to B becomes a *dead store*, and we can remove it.

LVN example (copy propagation, dead store elimination)

```
mov_l   vr20, $3          /* constant 3 */
mul_l   vr21, vr12, vr20  /* a * 3 */
mov_l   vr22, $4          /* constant 4 */
add_l   vr23, vr21, vr22  /* a * 3 + 4 */
mov_l   vr10, vr23        /* assign to x */
mov_l   vr24, vr20        /* constant 3 */
mov_l   vr25, vr21        /* a * 3 */
mov_l   vr26, $5          /* constant 5 */
sub_l   vr27, vr25, vr26  /* a * 3 - 5 */
mov_l   vr11, vr27        /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

Code after local value numbering; recomputations have been replaced by a `mov` to copy the originally-computed value.

LVN example (copy propagation, dead store elimination)

```
mov_l   vr20, $3          /* constant 3 */
mul_l   vr21, vr12, vr20  /* a * 3 */
mov_l   vr22, $4          /* constant 4 */
add_l   vr23, vr21, vr22  /* a * 3 + 4 */
mov_l   vr10, vr23        /* assign to x */
mov_l   vr24, vr20        /* constant 3 */
mov_l   vr25, vr21        /* a * 3 */
mov_l   vr26, $5          /* constant 5 */
sub_l   vr27, vr25, vr26  /* a * 3 - 5 */
mov_l   vr11, vr27        /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

Copy propagation: vr24 is a copy of vr20.

LVN example (copy propagation, dead store elimination)

```
mov_l   vr20, $3          /* constant 3 */
mul_l   vr21, vr12, vr20  /* a * 3 */
mov_l   vr22, $4          /* constant 4 */
add_l   vr23, vr21, vr22  /* a * 3 + 4 */
mov_l   vr10, vr23        /* assign to x */
mov_l   vr24, vr20        /* constant 3 */
mov_l   vr25, vr21        /* a * 3 */
mov_l   vr26, $5          /* constant 5 */
sub_l   vr27, vr25, vr26  /* a * 3 - 5 */
mov_l   vr11, vr27        /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

Copy propagation: vr25 is a copy of vr21.

LVN example (copy propagation, dead store elimination)

```
mov_l   vr20, $3          /* constant 3 */
mul_l   vr21, vr12, vr20  /* a * 3 */
mov_l   vr22, $4          /* constant 4 */
add_l   vr23, vr21, vr22  /* a * 3 + 4 */
mov_l   vr10, vr23        /* assign to x */
mov_l   vr24, vr20        /* constant 3 */
mov_l   vr25, vr21        /* a * 3 */
mov_l   vr26, $5          /* constant 5 */
sub_l   vr27, vr25, vr26  /* a * 3 - 5 */
mov_l   vr11, vr27        /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

Copy propagation: vr25 is used as a source operation by sub_l.

LVN example (copy propagation, dead store elimination)

```
mov_l   vr20, $3          /* constant 3 */
mul_l   vr21, vr12, vr20  /* a * 3 */
mov_l   vr22, $4          /* constant 4 */
add_l   vr23, vr21, vr22  /* a * 3 + 4 */
mov_l   vr10, vr23        /* assign to x */
mov_l   vr24, vr20        /* constant 3 */
mov_l   vr25, vr21        /* a * 3 */
mov_l   vr26, $5          /* constant 5 */
sub_l   vr27, vr21, vr26  /* a * 3 - 5 */
mov_l   vr11, vr27        /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

Copy propagation: replace vr25 with vr21.

LVN example (copy propagation, dead store elimination)

```
mov_l   vr20, $3          /* constant 3 */
mul_l   vr21, vr12, vr20  /* a * 3 */
mov_l   vr22, $4          /* constant 4 */
add_l   vr23, vr21, vr22  /* a * 3 + 4 */
mov_l   vr10, vr23        /* assign to x */
mov_l   vr24, vr20        /* constant 3 */
mov_l   vr25, vr21        /* a * 3 */
mov_l   vr26, $5          /* constant 5 */
sub_l   vr27, vr21, vr26  /* a * 3 - 5 */
mov_l   vr11, vr27        /* assign to y */
```

x is vr10
y is vr11
a is vr12
Temps start at vr20

Dead store elimination: there are no longer any uses of vr24 or vr25, so the instructions which assign to them can be removed.

Limits of local value numbering

Because it's a *local* analysis (scope is only one basic block), LVN can't exploit redundant computations that are in different basic blocks, even if the result of a computation is guaranteed to be available.

Superlocal value numbering

Superlocal value numbering extends LVN to propagate values known in a predecessor block to a successor block, but *only if the successor has no other predecessors*.

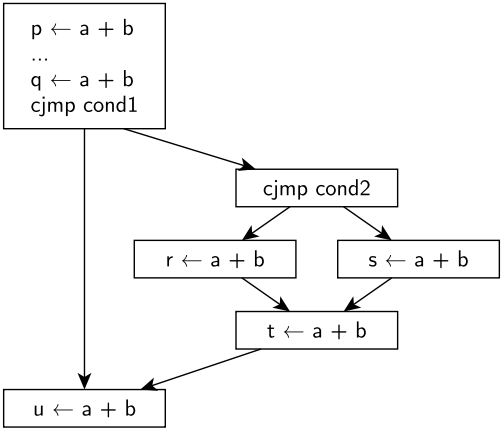
Superlocal value numbering example

```
p = a + b
...
q = a + b

if ( cond1 ) {
    if ( cond2 )
        r = a + b
    else
        s = a + b

    t = a + b
}

u = a + b
```



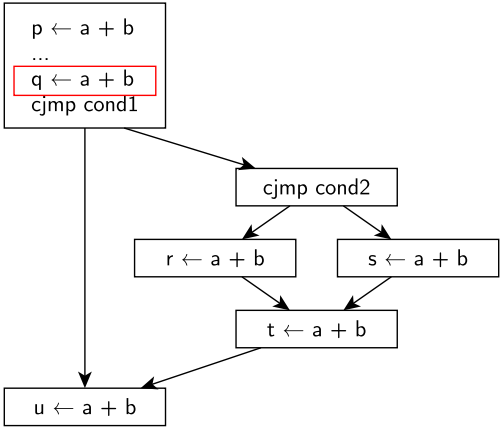
Superlocal value numbering example

```
p = a + b
...
q = a + b

if ( cond1 ) {
    if ( cond2 )
        r = a + b
    else
        s = a + b

    t = a + b
}

u = a + b
```



Redundancy fixable using LVN

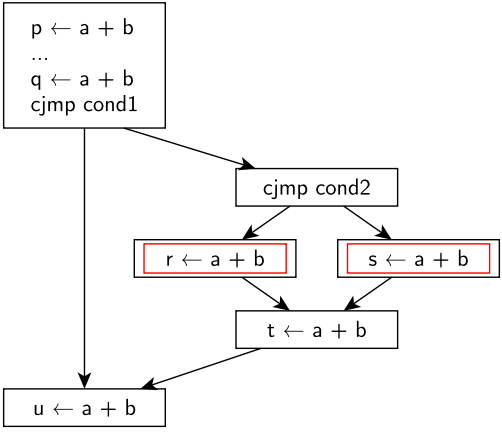
Superlocal value numbering example

```
p = a + b
...
q = a + b

if ( cond1 ) {
    if ( cond2 )
        r = a + b
    else
        s = a + b

    t = a + b
}

u = a + b
```



Additional redundancies fixable using super LVN

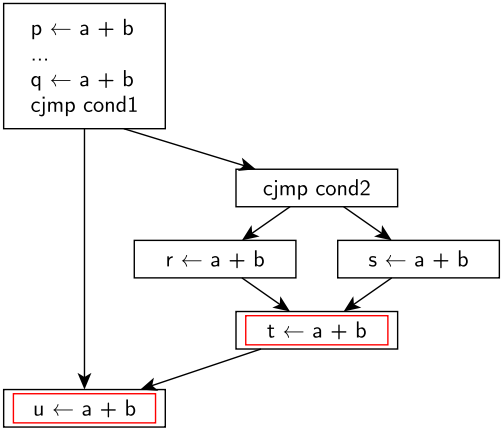
Superlocal value numbering example

```
p = a + b
...
q = a + b

if ( cond1 ) {
    if ( cond2 )
        r = a + b
    else
        s = a + b

    t = a + b
}

u = a + b
```



Redundancies requiring global techniques to fix

Implementing super LVN

To implement super LVN, the data structure keeping track of value numbers can simply be copied from the end of the predecessor block to the beginning of the (single) successor block.

Note that the register allocator will need to be involved: if a previously-computed value is in a register, and we want to reuse it in a different basic block, the register allocator will need to know (so it doesn't try to allocate that register to a different value.)

Implementing local value numbering

Implementing local value numbering

This section describes an approach to implementing LVN.

This is not the only possible approach.

LVN is a local analysis, so the analysis and transformation is applied separately to each basic block in the function.

Key concept: an *LVN key* represents a computed value.

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

- This is called *dead store elimination*

Local value numbering example

```
/* Note: vr10-vr14 are used for local vars */
localaddr vr16, $1600
sconv_lq vr17, vr11
mul_q    vr18, vr17, $80
add_q    vr19, vr16, vr18
sconv_lq vr20, vr12
mul_q    vr21, vr20, $8
add_q    vr22, vr19, vr21
mov_q    vr23, (vr22)
mov_q    vr15, vr23
localaddr vr24, $800
sconv_lq vr25, vr13
mul_q    vr26, vr25, $80
add_q    vr27, vr24, vr26
sconv_lq vr28, vr12
mul_q    vr29, vr28, $8
add_q    vr30, vr27, vr29
mov_q    vr32, (vr30)
mul_q    vr31, vr14, vr32
add_q    vr33, vr15, vr31
mov_q    vr15, vr33
```

```
/* Note: vr10-vr14 are used for local vars */
localaddr vr16, $1600
sconv_lq vr17, vr11
mul_q    vr18, vr17, $80
add_q    vr19, vr16, vr18
sconv_lq vr20, vr12
mul_q    vr21, vr20, $8
add_q    vr22, vr19, vr21
mov_q    vr23, (vr22)
mov_q    vr15, vr23
localaddr vr24, $800
sconv_lq vr25, vr13
mul_q    vr26, vr25, $80
add_q    vr27, vr24, vr26
sconv_lq vr28, vr12
mul_q    vr29, vr28, $8
add_q    vr30, vr27, vr29
mov_q    vr32, (vr30)
mul_q    vr31, vr14, vr32
add_q    vr33, vr15, vr31
mov_q    vr15, vr33
```