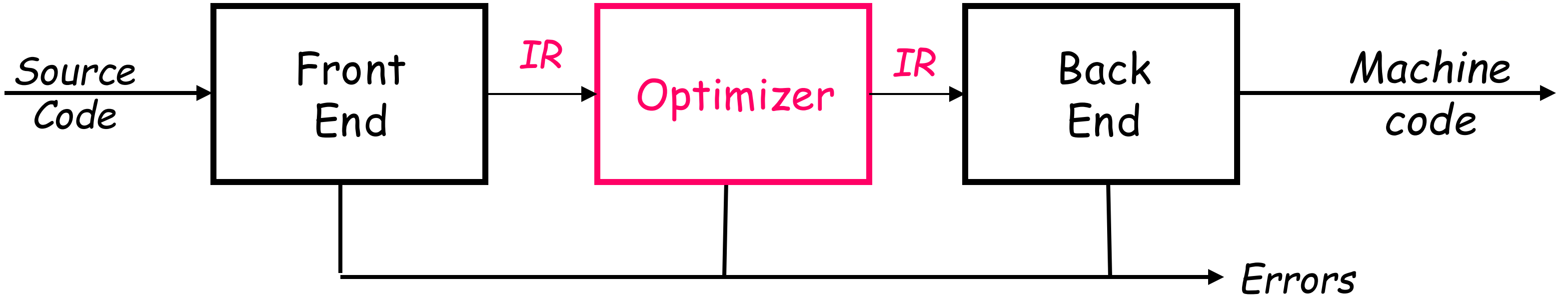
COMP 412
FALL 2010

# *Introduction to Code Optimization*

## *Comp 412*

This lecture begins the material from Chapter 8 of EaC

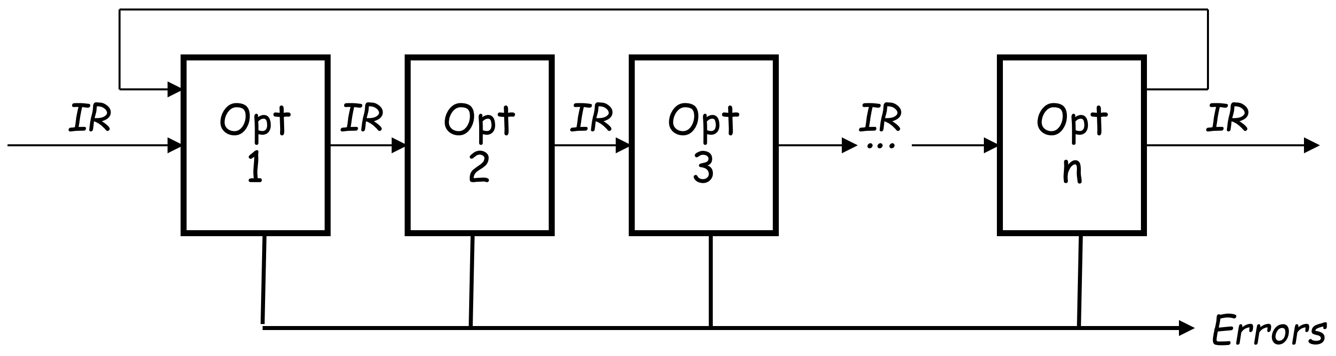
# Traditional Three-Phase Compiler




## Optimization (or Code Improvement)

- Analyzes IR and rewrites (or *transforms*) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, …
- Must preserve "meaning" of the code
  - Measured by values of named variables
  - A course (or two) unto itself

# The Optimizer




*Modern optimizers are structured as a series of passes*

## Typical Transformations

- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form

# The Role of the Optimizer

- The compiler can implement a procedure in many ways
- The optimizer tries to find an implementation that is "better"
  - *Speed, code size, data space, …*

To accomplish this, it

- Analyzes the code to derive knowledge about run-time behavior
  - Data-flow analysis, pointer disambiguation, …
  - General term is "static analysis"
- Uses that knowledge in an attempt to improve the code
  - Literally hundreds of transformations have been proposed
  - Large amount of overlap between them

Nothing "optimal" about optimization

- Proofs of optimality assume restrictive & unrealistic conditions

# Redundancy Elimination as an Example

An expression x+y is redundant if and only if, along every path from the procedure's entry, it has been evaluated, and its constituent subexpressions (x & y) have not been re-defined.

If the compiler can prove that an expression is redundant

- It can preserve the results of earlier evaluations
- It can replace the current evaluation with a reference

Two pieces to the problem

- Proving that x+y is redundant, or *available*
- Rewriting the code to eliminate the redundant evaluation

One technique for accomplishing both is called *value numbering*

Assume a low-level, linear IR such as ILOC

# Value Numbering

Local algorithm due to Balke (1968) or Ershov (1954)

The key notion
- Assign an identifying number, V(n), to each expression
  - V(x+y) = V(j) iff x+y and j always have the same value
  - Use hashing over the value numbers to make it efficient
- Use these numbers to improve the code

Within a basic block; definition becomes more complex across blocks

Improving the code
- Replace redundant expressions
  - Same VN ⇒ refer rather than recompute
- Simplify algebraic identities
- Discover constant-valued expressions, fold & propagate them
- Technique designed for low-level, linear IRs, similar methods exist for trees (e.g., build a DAG)

# Local Value Numbering

## The Algorithm

For each operation $o$ = ‹operator, $o_1$, $o_2$› in the block, in order

1. Get value numbers for operands from hash lookup
2. Hash ‹operator,VN($o_1$),VN($o_2$)› to get a value number for *o*
3. If *o* already had a value number, replace *o* with a reference
4. If $o_1$ & $o_2$ are constant, evaluate it & replace with a loadI

If hashing behaves, the algorithm runs in linear time

- If not, use multi-set discrimination† or acyclic DFAs††

Handling algebraic identities

- Case statement on operator type
- Handle special cases within each operator

†*see p. 251 in EaC*

††DFAs for REs without closure can be built online to provide a "perfect hash"

# Local Value Numbering

## An example

| Original Code | With VNs | Rewritten |
|---|---|---|
| $a \leftarrow x + y$ | $a^3 \leftarrow x^1 + y^2$ | $a^3 \leftarrow x^1 + y^2$ |
| * $b \leftarrow x + y$ | * $b^3 \leftarrow x^1 + y^2$ | * $b^3 \leftarrow a^3$ |
| $a \leftarrow 17$ | $a^4 \leftarrow 17$ | $a^4 \leftarrow 17$ |
| * $c \leftarrow x + y$ | * $c^3 \leftarrow x^1 + y^2$ | * $c^3 \leftarrow a^3$ (oops!) |

*Two redundancies*

- Eliminate stmts with a *
- Coalesce results ?

*Options*

- Use $c^3 \leftarrow b^3$
- Save $a^3$ in $t^3$
- Rename around it

# Local Value Numbering

Example (continued):

| Original Code | With VNs | Rewritten |
|---|---|---|
| $a_0 \leftarrow x_0 + y_0$ | $a_0^3 \leftarrow x_0^1 + y_0^2$ | $a_0^3 \leftarrow x_0^1 + y_0^2$ |
| $*\ b_0 \leftarrow x_0 + y_0$ | $*\ b_0^3 \leftarrow x_0^1 + y_0^2$ | $*\ b_0^3 \leftarrow a_0^3$ |
| $a_1 \leftarrow 17$ | $a_1^4 \leftarrow 17$ | $a_1^4 \leftarrow 17$ |
| $*\ c_0 \leftarrow x_0 + y_0$ | $*\ c_0^3 \leftarrow x_0^1 + y_0^2$ | $*\ c_0^3 \leftarrow a_0^3$ |

Renaming:
- Give each value a unique name
- Makes it clear

Notation:
- While complex, the meaning is clear

Result:
- $a_0^3$ is available
- Rewriting now works

# Local Value Numbering

## The Algorithm

For each operation *o* = <operator, $o_1$, $o_2$> in the block, in order

1 Get value numbers for operands from hash lookup

2 Hash <operator,VN($o_1$),VN($o_2$)> to get a value number for *o*

3 If *o* already had a value number, replace *o* with a reference

4 If $o_1$ & $o_2$ are constant, evaluate it & replace with a loadI

asymptotic

constants

## Complexity & Speed Issues

- "Get value numbers" — linear search versus hash
- "Hash <op,VN($o_1$),VN($o_2$)>" — linear search versus hash
- Copy folding — set value number of result
- Commutative ops — double hash versus sorting the operands

# Simple Extensions to Value Numbering

## Constant folding

- Add a bit that records when a value is constant
- Evaluate constant values at compile-time
- Replace with load immediate or immediate operand
- No stronger local algorithm

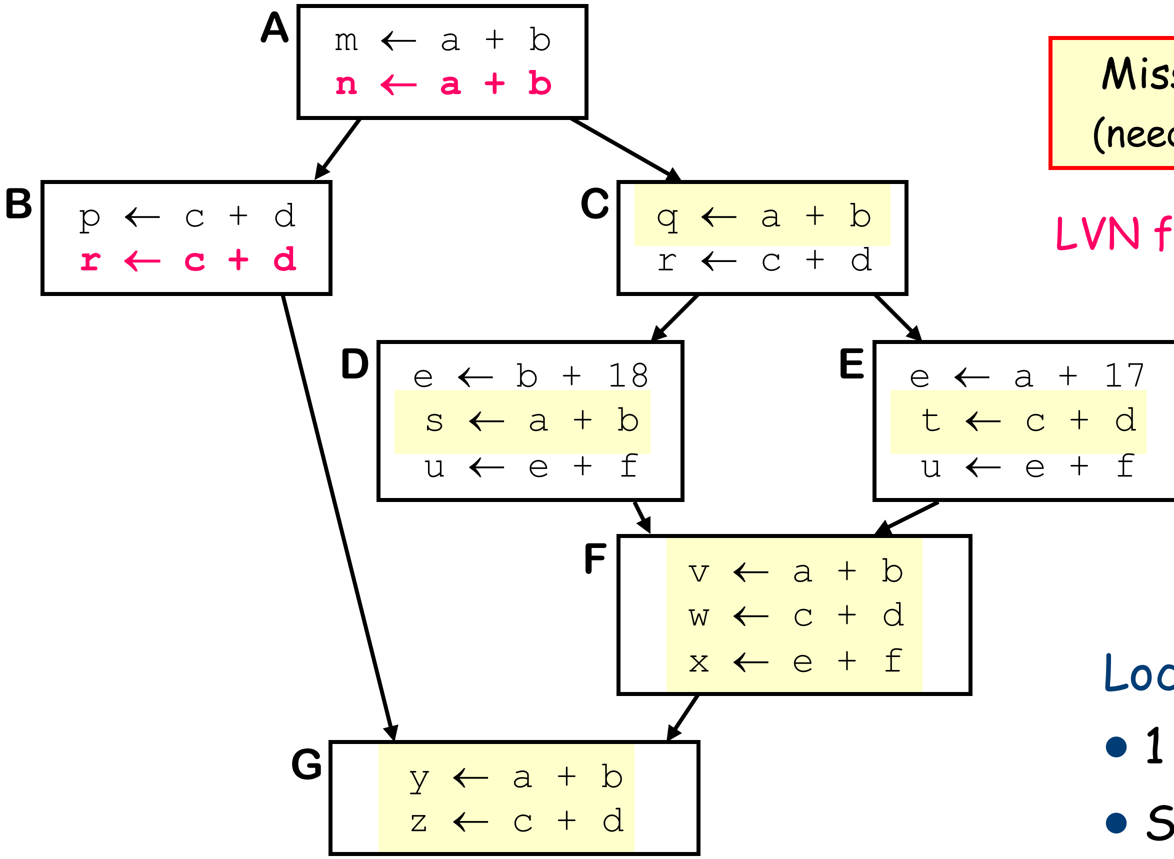
Identities (on VNs)

$x \leftarrow y$, $x+0$, $x-0$, $x*1$, $x \div 1$, $x-x$, $x*0$, $x \div x$, $x \vee 0$, $x \wedge$ 0xFF…FF, max(x,MAXINT), min(x,MININT), max(x,x), min(y,y), and so on ...

## Algebraic identities

- Must check (many) special cases
- Replace result with input VN
- Build a decision tree on operation

# Value Numbering




Missed opportunities
(need stronger methods)

LVN finds these redundant ops

Local Value Numbering

- 1 block at a time
- Strong local results
- No cross-block effects