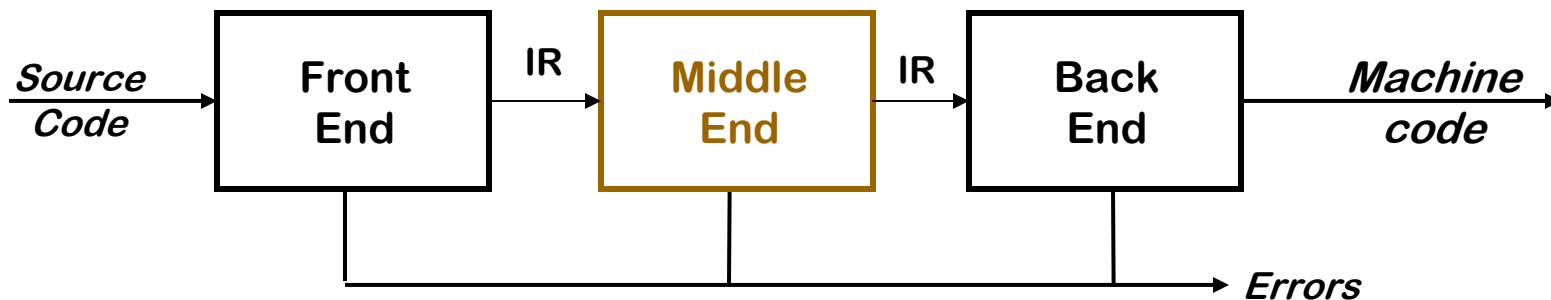


Introduction to Optimization

Copyright 2003, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

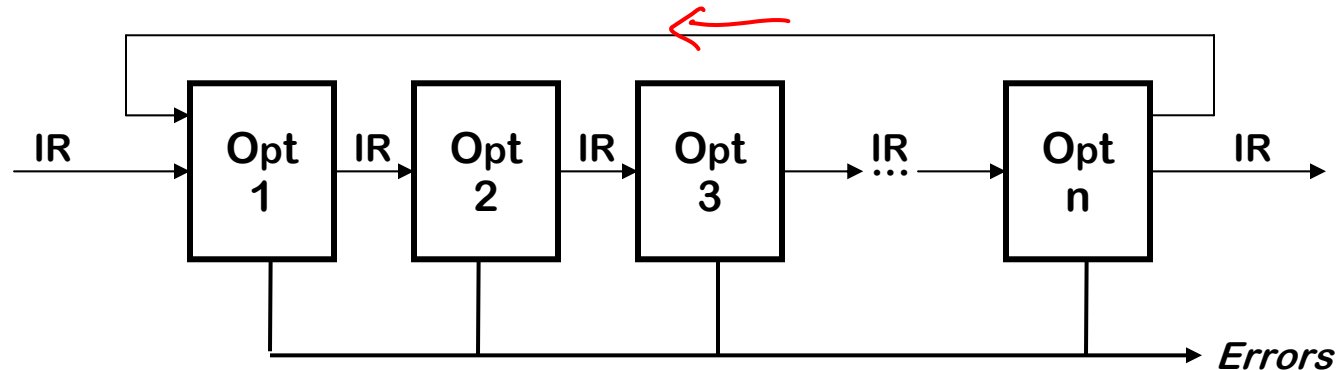
Traditional Three-pass Compiler



Code Improvement (or Optimization)

- 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 (or Middle End)



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 *analyze*
 - Data-flow analysis, pointer disambiguation, ...
 - General term is "static analysis"
- Uses that knowledge in an attempt to improve the code *transform*
 - 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
- Rewriting the code to eliminate the redundant evaluation

One technique for accomplishing both is called value numbering

Value Numbering

(An old idea)

The key notion

(Balke 1968 or Ershov 1954)

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

Improving the code

- Replace redundant expressions
- Simplify algebraic identities
- Discover constant-valued expressions, fold & propagate them
- This technique was invented for low-level, linear IRs
- Equivalent methods exist for trees *(build a DAG)*

Local Value Numbering

The Algorithm

For each operation $o = \langle \text{operator}, o_1, o_2 \rangle$ in the block

- 1 Get value numbers for operands from hash lookup
- 2 Hash $\langle \text{operator}, \text{VN}(o_1), \text{VN}(o_2) \rangle$ 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 load

— from beginning to end
— assign VNs if not already done

If hashing behaves, the algorithm runs in linear time

→ If not, use multi-set discrimination (see digression in Ch. 5)

Handling algebraic identities

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

Local Value Numbering

An example

Original Code

```

a ← x + y
* b ← x + y
a ← 17
* c ← x + y
    
```

With VNs

```

a3 ← x1 + y2
* b3 ← x1 + y2 redundant
a4 ← 174
* c3 ← x1 + y2 redundant
    
```

Rewritten

enter x 1

```

a3 ← x1 + y2
* b3 ← a3
a4 ← 173
* c3 ← a3 (4ops!)
    
```

a 4

Two redundancies:

- Eliminate stmts with a *
- Coalesce results ?

Options:

- Use c³ ← ~~b³~~
- Save a³ in t³
- Rename around it

Local Value Numbering

Example (continued)

Original Code

$a_0 \leftarrow x_0 + y_0$
* $b_0 \leftarrow x_0 + y_0$
 $a_1 \leftarrow 17$
* $c_0 \leftarrow x_0 + y_0$

With VNs

$a_0^3 \leftarrow x_0^1 + y_0^2$
* $b_0^3 \leftarrow x_0^1 + y_0^2$
 $a_1^4 \leftarrow 17$
* $c_0^3 \leftarrow x_0^1 + y_0^2$

Rewritten

$a_0^3 \leftarrow x_0^1 + y_0^2$
* $b_0^3 \leftarrow a_0^3$
 $\rightarrow a_1^4 \leftarrow 17$
* $c_0^3 \leftarrow a_0^3$

Renaming:

- Give each value a unique name
- Makes it clear

Notation:

- While complex, the meaning is clear

Result:

- a_0^3 is available
- Rewriting just works

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

Algebraic identities

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

Identities: (Click)
 $x \leftarrow y$, $x+0$, $x-0$, $x*1$, $x\div 1$, $x-x$,
 $x*0$, $x\div x$, $x\vee 0$, $x \wedge 0xFF\dots FF$,
 $\max(x, \text{MAXINT})$, $\min(x, \text{MININT})$,
 $\max(x, x)$, $\min(y, y)$, and so on ...

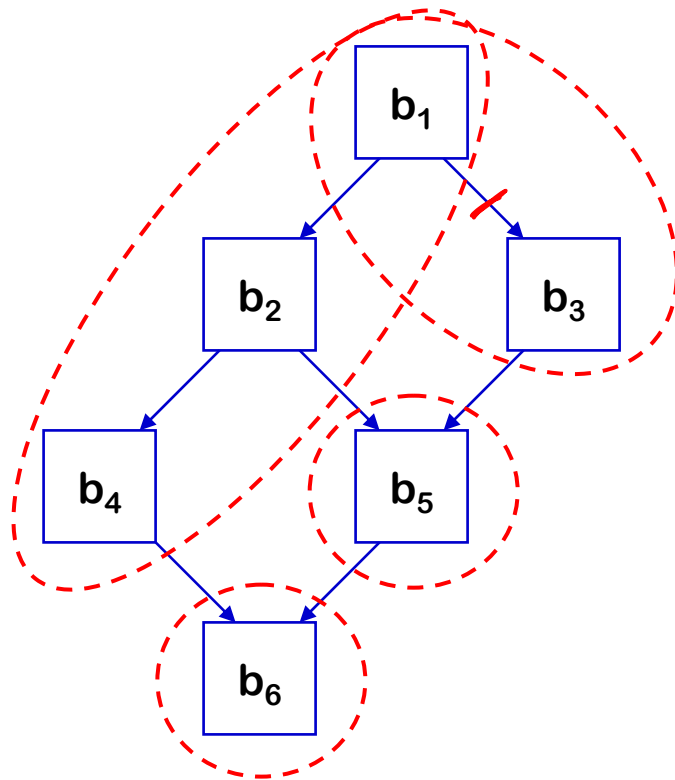
With values, not names

Handling Larger Scopes

Extended Basic Blocks

- Initialize table for b_i with table from b_{i-1}
- With single-assignment naming, can use scoped hash table

Otherwise, it is complex



The Plan:

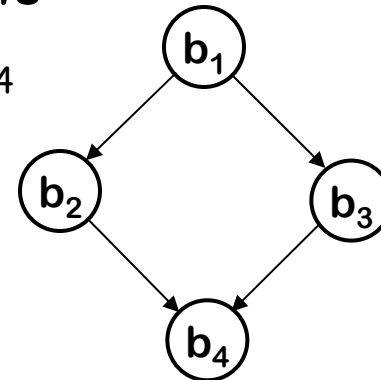
- Process b_1, b_2, b_4
Pop two levels
- Process b_3 relative to b_1
- Start clean with b_5
- Start clean with b_6
-

Using a scoped table makes doing the full tree of EBBs that share a common header efficient.

Handling Larger Scopes

To go further, we must deal with merge points

- Our simple naming scheme falls apart in b_4
- We need more powerful analysis tools
- Naming scheme becomes SSA



This requires global data-flow analysis

“Compile-time reasoning about the run-time flow of values”

- 1 Build a model of control-flow
- 2 Pose questions as sets of simultaneous equations
- 3 Solve the equations
- 4 Use solution to transform the code

Examples: **LIVE, REACHES, AVAIL**