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
**Why are optimizers needed?**

**Reduce programmer effort**
- automatically generate efficient code
- less work for programmer
- below “optimal” hand-optimized code

**Undo high-level abstractions**
- some optimizations not possible for language
- flatten control flow to branches
- convert method lookups to subroutine calls
- map data structures to addresses

**Maintain performance portability**
- performance depends on architecture
- optimizations by programmer too specific
- compiler can customize program for processor

---

**Code optimizations**

**Reduce execution time**
- historically, to avoid assembly coding
- support higher levels of abstraction
- support more complex processors
- important applications: science, databases

**Reduce space**
- historically, small expensive memories may trade space for speed
- space may reduce speed (caches)
- new areas: internet applets, embedded processors

**Level of optimization**
- source code
- intermediate representation
- binary machine code
- at run-time
**Code optimization**

How can optimizations improve code quality?

**Machine-independent transformations**
- remove unnecessary computations
- simplify control structures
- move code to a less frequently executed place
- specialize some general purpose code
- find useless code and remove it
- expose opportunities (enable) for other optimizations

**Machine-dependent transformations**
- replace complex operation with simpler one
- exploit special instructions
- exploit memory hierarchy (registers, cache)

---

**Types of optimization**

- classical - reduce the number/cost of instructions execute
- register allocation - keep values in registers, eliminate loads/stores
- instruction scheduling - hide instruction latency, exploit instruction-level parallelism
- data locality - keep data accesses in faster levels of memory hierarchy (registers, cache, TLB, memory)
- Multiprocessing - compute in parallel on multiple processors

**Optimization framework**

- ideally, maintain separation of concerns
- in practice, integrate optimization algorithms
**Code optimization**

Three considerations arise in applying a transformation.

- Safety - Does applying the transformation change the results of executing the code?
- Profitability - Is there a reasonable expectation that applying the transformation will improve the code?
- Opportunity - Can we efficiently and frequently find places to apply optimization?

Need a clear understanding of these issues.

Profitability is particularly tricky...

Learn how the compiler decides when transformations will be applicable, safe, and profitable.

---

**Classical transformation examples**

- Assumption: Anything executed once takes no time. Gains achieved by looping at program loops.
- Break program into straightline code segments, bounded by branches (**Basic Blocks**)  
- Connect basic blocks together into a program graph  
- Find innermost loop  
- Find loop invariant expressions and move them to head of loop  
- Find common subexpressions and combine them  
- Move loop invariant expressions out of loop  
- Repeat process with next outer loop
Classical Transformation Examples

• Unreachable code - eliminate code not reached during program execution
  goto L:
  unreachable code
  L:                                                (Delete all this code)

• Control-flow simplification - remove jumps to jumps by analyzing targets of jumps
  goto L      (goto M)
  code
  L: goto M
  code
  M:

• Algebraic simplification - simplify arithmetic expressions by analyzing expression trees
  A := 0
  C := B + A                      (C:= B)

• Constant folding - replace constant expressions with result
  A := 5
  B := 6
  C := B + A                      (C:=11)

• Idiom recognition - replace operations with less expensive idioms
  B := A * 16  (Shift right)
  D := B / 4

• Available expressions - reuse values always available
  C := B + A
  D := B + A                      (D:=C)

• Dead code elimination - eliminate unnecessary computations
  A := 5                      (Delete this statement)
  A := 6

• Copy propagation - propagate names into copy instructions
  B := A                      (Delete this statement)
  C := B                      (C:=A)
Classical Transformation Examples

- **Procedure integration**
  call S(A,B)  \textit{(expand as inline code)}

- **Loop unrolling**
  for \( l := 1 \) to \( 100 \);
  end;

  becomes
  for \( l := 1 \) to \( 100 \) by \( 2 \);
  end;

  increases opportunity for parallelism (multiple processors or pipelining) but increases program size

- **Jamming** -- opposite of loop unrolling
  for \( l := 1 \) to \( 100 \);
  end;

  for \( l := 1 \) to \( 100 \);
  \[ C[l] := C[l] + D[l]; \]
  end;

  becomes
  for \( l := 1 \) to \( 100 \);
  \[ C[l] := C[l] + D[l]; \]
  end;

  increases opportunity for parallelism (multiple processors or pipelining)

- **Common subexpression elimination**
  \[ A := B+C; \]
  \[ D := B+C; \]

  becomes
  \[ A := B+C; \]
  \[ D := A; \]

  reduces size of program, reduces execution time and can increase execution time of program (How?)
Classical Transformation Examples

- Code motion - reduces execution of redundant instructions

  ```
  while X<Y do
  A := B+C;
  X:= X+1;
  end;
  becomes
  A := B+C;
  while X<Y do
  X:= X+1;
  end;
  ```

- Basic blocks -- Fundamental concept for all code improvement algorithms --
  sequence of code where control enters at top, exits at bottom,
  no branch/halt except at end

- Construction algorithm (for 3-address code)
  - determine set of leaders
  - first statement
  - target of goto or conditional goto
  - statement following goto or conditional goto

- Add to basic block all statements following leader up to next leader or end of
  program

  ```
  A := 0 (Block 1)
  if (<cond>) goto L
  A := 1 (Block 2)
  B := 1 (Block 2)
  L:   C := A (Block 3)
  ```

Scope of optimizations

Scope

- peephole --- across a few instructions
- local --- within basic block
- global --- across basic blocks
- interprocedural --- across procedures

Some optimizations may be applied locally or globally (e.g., dead code elimination):

```
A := 0  A := 0
A := 1  if (<cond>) goto L
B := A  A := 1
B := A
```

Some optimizations require global analysis (e.g., loop-invariant code motion):

```
while (<cond>) do
  A := B + C
  foo(A)
end
```
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

General optimization process

• Generate graph of program, based on basic blocks
  1. Compute live/dead analysis for all variables
  2. Redundancy

• Look for common subexpressions using liveness analysis to determine if variable values have been changed to see if two expressions have the same value
1. Live/dead variable analysis

- Determine path through a program where a variable’s value does not change
- If a and b do not change, then the expression (a+b) anywhere along this path contains the same value
  > Can compute (a+b) once and use computed value for each occurrence.
- If two variables (e.g., temporaries) are not used along the same path, they can share the same memory location or register
  > Better use of registers
  > Less storage to use

1. Definitions

- **Live** – a variable is live if its value will be used before the variable is redefined
- **Def** – The def of a variable is the set of graph nodes that define a value to that variable
- **Use** – The use of a variable is the set of graph nodes that access the value of a variable

- Def and Use vectors are a syntactic property of a program.
- Give the graph structure and the Def and Use vectors at each node of a program graph, you can compute the liveness of each variable.
1. Example liveness property

- B is live 2→4
- C is live on entry,
  - Live 1→3,
  - Live 3→3,
  - Live 3→6
- A is live 1→2
  - Live 4→2
- Example:
  - Def(3) = {C}
  - Use(3) = {B, C}
  - Live(3) = {B, C}

```
A := 0
B := A + 1
C := C + B
A := 2 * B
A < N
Return C
```

1. Computing liveness

- Live-in(n) – Variables live as input to block n
- Live-out(n) – Variables live as output from block n

- in(n) = use(n) ∪ (out(n) – def(n))
- out(n) = ∪ in(s) for s ∈ succ(n)

Solving this set of equations for all n gives the liveness property for each variable (e.g., see page 225 of text).

- Time to compute is: O(N^4) as worst case
- Usual time is O(N) to O(N^2)
2. 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**

---

2. Data Flow Equations for Availability

- Constants
  - $\text{DEF}(b)$ — subexpressions defined in $b$ and available on exit
  - $\text{NOTKILLED}(b)$ — subexpressions that are not killed in $b$
    - A subexpression is killed if either input is assigned to

- Computing $\text{AVAIL}(b)$ — the set of expressions available on input to block $b$

$$\text{AVAIL}(b) = \cap_{p \in \text{Pred}(b)} (\text{DEF}(p) \cup (\text{AVAIL}(p) \cap \text{NOTKILLED}(p)))$$

- What is the starting value for $\text{AVAIL}(b)$?
- What is the problem with this formulation?
2. Value Numbering A 1960’s 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 \( \forall \) inputs
  - Use hashing over 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)

2. Local Value Numbering

The algorithm

For each operation \( o = <\text{operator}, o_1, o_2> \) in the block
1. Get value numbers for operands from hash lookup
2. Hash \( <\text{operator}, VN(o_1), VN(o_2)> \) 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 load!

If hashing behaves, the algorithm runs in linear time
  - If not, try multi-set discrimination

Handling algebraic identities

- Case statement on operator type
- Handle special cases within each operator
2. Local Value Numbering

An example

<table>
<thead>
<tr>
<th>Original Code</th>
<th>With VNs</th>
<th>Rewritten</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a \leftarrow x + y)</td>
<td>(a_3 \leftarrow x^1 + y^2)</td>
<td>(a^3 \leftarrow x^1 + y^2)</td>
</tr>
<tr>
<td>(* \ b \leftarrow x + y)</td>
<td>(b^3 \leftarrow x^1 + y^2)</td>
<td>(* \ b^2 \leftarrow a^3)</td>
</tr>
<tr>
<td>(a \leftarrow 17)</td>
<td>(a^4 \leftarrow 17)</td>
<td>(a^4 \leftarrow 17)</td>
</tr>
<tr>
<td>(* \ c \leftarrow x + y)</td>
<td>(c^3 \leftarrow x^1 + y^2)</td>
<td>(* \ c^3 \leftarrow a^3) (oops!)</td>
</tr>
</tbody>
</table>

Two redundancies:
- Eliminate stmts with \(a\) *
- Coalesce results?

Options:
- Use \(c^3 \leftarrow b^3\)
- Save \(a^3\) in \(t^3\)
- Rename around it

2. Local Value Numbering

Example (continued)

<table>
<thead>
<tr>
<th>Original Code</th>
<th>With VNs</th>
<th>Rewritten</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_0 \leftarrow x_0 + y_0)</td>
<td>(a_0^3 \leftarrow x_0^1 + y_0^2)</td>
<td>(a_0^3 \leftarrow x_0^1 + y_0^2)</td>
</tr>
<tr>
<td>(* \ b_0 \leftarrow x_0 + y_0)</td>
<td>(b_0^3 \leftarrow x_0^1 + y_0^2)</td>
<td>(* \ b_0^3 \leftarrow a_0^3)</td>
</tr>
<tr>
<td>(a_1 \leftarrow 17)</td>
<td>(a_1^4 \leftarrow 17)</td>
<td>(a_1^4 \leftarrow 17)</td>
</tr>
<tr>
<td>(* \ c_0 \leftarrow x_0 + y_0)</td>
<td>(c_0^3 \leftarrow x_0^1 + y_0^2)</td>
<td>(* \ c_0^3 \leftarrow a_0^3)</td>
</tr>
</tbody>
</table>

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:
- $x \leftarrow y$, $x + 0$, $x - 0$, $x \cdot 1$, $x + 1$, $x \cdot x$, $x \cdot 0$,
- $x \cdot x$, $x \cdot 0$, $x \cdot 0xFF...FF$,
- max($x$, MAXINT), min($x$, MININT), max($x$, $x$), min($y$, $y$), and so on ...

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

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 b₄
- 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

Constant Propagation

- **Goal:** Produce an algorithm that will propagate all constants in a procedure, replacing constant expressions with the result of evaluating the expression at compile time
- **Strategy:**
  - Construct a graph that maps definitions to uses within a procedure — def-use chains
  - Propagate constants forward from points of constant definitions along def-use chains
  - Evaluate new constant expressions whenever they are identified
  - Stop when no more constants are available
- **Challenges**
  - Constructing def-use chains
  - Identifying constant expressions
**Constructing Def-Use Chains**

- Perform REACHES calculation
  \[
  \text{REACHES}(b) = \bigcup_{p \in \text{pred}(b)} \text{REACHESOUT}(p)
  \]
  \[
  \text{REACHESOUT}(b) = \text{DEFSOUT}(b) \cup (\text{REACHES}(b) \cap \text{NOREDEF}(b))
  \]

- At each use of variable \( x \), construct a DEF-USE chain to \( x \) from each definition \( y \) in REACHES at \( x \)

- Note: REACHES sets easy to propagate forward in basic blocks

---

**Dead Code Elimination**

\[
\text{worklist} := \{\text{absolutely useful statements}\};
\]

\[
\text{while worklist} \neq \emptyset \text{ do begin}
\]

\[
\text{worklist} := \text{worklist} - \{x\};
\]

\[
\text{mark } x \text{ useful;}
\]

\[
\text{for all } (y,x) \in \text{DefUse do}
\]

\[
\text{if } y \text{ is not marked useful then worklist} := \text{worklist} \cup \{y\};
\]

\[
\text{end}
\]

\[
\text{delete every statement that is not marked useful;}
\]
**Constant Propagation Lattice**

```
unknown

• -3    -2    -1    0     1     2     3

non-constant
```

**Constant Propagation Algorithm**

```plaintext
for all statements s in the program do begin
  for each output v of s do valout(v,s) := unknown;
  for each input w of s do
    if w is a variable then valin(w,s) := unknown;
    else valin(w,s) := the constant value of w;
end

worklist := {all statements of constant form, e.g., X = 5};
while worklist ≠ ∅ do begin
  choose and remove an arbitrary statement x from worklist;
  let v denote the output variable for x;
  newval := m(x)(valin(v,x), for all inputs v to x);
  if newval ≠ valout(v,x) then begin
    valout(v,x) := newval;
    for all (x,y) DefUse do begin
      oldval := valin(v,y);
      valin(v,y) := oldval, valout(v,x);
      if valin(v,y) ≠ oldval then worklist := worklist {y};
    end
  end
end
```
Advantages and Disadvantages

- Advantages
  - Linear in the size of the Def-Use graph
    - Why?

- Disadvantage
  - Def-Use graph could be large

Shrinking the Graph: SSA

- Static Single-Assignment Form

At most one definition reaches each use!
**Constructing SSA**

- Find points of insertion for $\phi$-functions and insert them
  - Where should they go?

![Diagram]

Put a $\phi$-function for $x$ in every block in the dominance frontier for block $b$.

The **dominance frontier** $DF(b)$ for a given block $b$ is the set of blocks $q$ such that some predecessor of $q$ is dominated in the control-flow graph by $b$, but $q$ itself is not strictly dominated by $b$.

**Dominator**s

- A node $x$ in directed graph $G$ with a single exit node **predominates** (or **dominates**) node $y$ in $G$ if any path from the entry node of $G$ to $y$ must pass through $x$.

- The **immediate dominator** of a block $x$ is the block $y$ in dominators($x$) such that dominators($y$) = dominators($x$) - {$x$}

- How do we compute dominators($b$)?

\[ \text{DOMINATORS}(b) = \{b\} \cup \bigcap_{p \in \text{preds}(b)} \text{DOMINATORS}(p) \]

- Is it really this easy?
Computing Dominance Frontiers

Find the immediate dominator relation idom for the control-flow graph G; (For a control-flow graph with a single entry, this relation forms a tree, with the entry node as the root.)

Let I be a topological listing of the dominator tree such that, if x dominates y, then x comes after y in I;

while I ≠ ∅ do begin
  let x be the first element of I;
  remove x from I;
  for all control flow successors y of x do
    if idom(y) ≠ x then DF(x) = DF(x) \{y\};
  for all z such that idom(z) = x do
    for all y in DF(z) do
      if idom(y) ≠ x then DF(x) = DF(x) \{y\};
end

Algorithms on SSA

• Dead code elimination and constant propagation work unchanged, assuming a meaning is constructed for \(\phi\)-functions;
  > The edge set should be much smaller, so the algorithms should run faster
• Many other algorithms can exploit the single-assignment property
  > What about value numbering?
  > Since each value has a unique name, you can do value number on SSA in complex control flow