Introduction to Optimization

Local Value Numbering

Copyright 2019, Keith D. Cooper, Linda Torczon & Zoran Budimlić, all rights reserved.

Students enrolled in Comp 506 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.
Optimization

Compilers operate at multiple *granularities* or *scopes*

- **Local** techniques
  - Work on a single basic block
  - Maximal length sequence of straight-line code

- **Regional** techniques
  - Consider multiple blocks, but less than whole procedure
  - Single loop, loop nest, dominator region, ...

- **Intraprocedural**, or **global**, techniques
  - Operate on an entire procedure
  - Common unit of compilation

- **Interprocedural**, or **whole-program**, techniques
  - Operate on > 1 procedure, up to whole program
  - Logistical issues related to accessing the code

Content from §8.2 EaC2e
At each of these scopes, the compiler uses different graphs

- **Local** techniques
  - Dependence graph *(instruction scheduling)*
  - Expression trees *(tree-height balancing)*

- **Regional** Techniques
  - Control-flow graph *(CFG)* *(natural loops)*
  - Dominator tree

- **Intraprocedural**, or **global**, techniques
  - Control-flow graph
  - Def-Use chains, sparse evaluation graphs, graph of SSA connections

- **Interprocedural**, or **whole-program**, techniques
  - Call (multi) graph

**Compiler writers should be able to perform graph traversals in their sleep** – preorder, postorder, reverse postorder, depth first, ...
Optimization

At each of these scopes, the techniques take different approaches

• **Local** techniques
  ♦ Simple in-order walks of the block

• **Regional** Techniques
  ♦ Find a way to treat multiple blocks as a single block
  ♦ Work with an entire loop nest

(EBBs, dominators)

• **Intraprocedural**, or **global**, techniques
  ♦ Data-flow analysis to determine safety and opportunity
  ♦ Separate transformation phase to rewrite the code

• **Interprocedural**, or **whole-program**, techniques
  ♦ Need a compilation framework where optimizer can see all the relevant code
  ♦ Sometimes, limit to all procedures in a file
  ♦ Sometimes, perform optimization at link time
Optimization

We want to differentiate between analysis and transformation

• Analysis reasons about the code’s behavior
• Transformation rewrites the code to change its behavior

Each needs the other to be worthwhile

Local techniques can interleave analysis and transformation

• The most important property of a basic block is that the operations execute in a predictable, defined order

In larger scopes, the compiler typically must complete its analysis before it transforms the code

• Analysis must consider all possible paths, including cycles
  ♦ Cycles and if-then-else branches typically force compiler into offline analysis
• Leads to confusion in terminology between “optimization”, “analysis”, and “transformation”
Optimization

Terminology

**Optimization**

- We will use “optimization” to refer to a broad technique or strategy, such as code motion or dead code elimination

**Transformation**

- We will use “transformation” to refer to algorithms & techniques that rewrite the code being compiled

**Analysis**

- We will use the term “analysis” to refer to algorithms & techniques that derive information about the code being compiled

This subtle distinction in usage was suggested by Vivek Sarkar, now at Georgia Tech.
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 instance of 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

**Value numbering** is a single-pass, local technique that accomplishes both.
Local Value Numbering

The first step in performing local value numbering is to identify the basic blocks — maximal length sequences of straight-line (branch-free) code

• Find all the basic blocks in the code
• Apply local value numbering to each block

To find the basic blocks, we build a control-flow graph

• Nodes represent basic blocks
• Edges represent the flow of control

Once the compiler has a CFG, it can value number each block
Building a Control-Flow Graph

The first step in almost any optimization is building a CFG

// find all the leaders, assume first op
// & block are numbered zero
next ← 0       // block number
leader[next++] ← 0 // block’s 1st op

// find other leaders
for i ← 0 to n
    if op[i] is a jump
        then leader[next++] ← target(i)
    if op[i] is a branch then
        leader[next++] ← taken(i)
        leader[next++] ← not_taken(i)

// build all the blocks
for i ← 0 to next − 1
    j ← leader[i] + 1
    while j ≤ n and j ∉ leader
        j ← j + 1
    last[i] ← j − 1

If target, taken, or not_taken are ambiguous, then we must include all labeled ops as leaders.

Sources of ambiguous targets:
- Jump or branch to a register
- PC-relative jump or branch

No Ambiguity In ILOC:
All branches in ILOC have two explicit targets. Branches and jumps target a label rather than a register.

In the original compiler, jump to register was followed with an advisory list of labels generated when the ILOC was generated.

§ 5.3.4 in EaC2e (p. 241)
Building a Control-Flow Graph

The first step in almost any optimization is building a CFG

// find all the leaders, assume first op & block are numbered zero
next ← 0 // block number
leader[next++] ← 0 // block’s 1st op

// find other leaders
for i ← 0 to n
    if op[i] is a jump
        then leader[next++] ← target(i)
    if op[i] is a branch then
        leader[next++] ← taken(i)
        leader[next++] ← not_taken(i)

// build all the blocks
for i ← 0 to next – 1
    j ← leader[i] + 1
    while j ≤ n and j ∉ leader
        j ← j + 1
    last[i] ← j – 1

// find all the leaders, assume first op & block are numbered zero
next ← 0 // block number
leader[next++] ← 0 // block’s 1st op

// find other leaders
for i ← 0 to n
    if op[i] is a jump
        then leader[next++] ← target(i)
    if op[i] is a branch then
        leader[next++] ← taken(i)
        leader[next++] ← not_taken(i)

// build all the blocks
for i ← 0 to next – 1
    j ← leader[i] + 1
    while j ≤ n and j ∉ leader
        j ← j + 1
    last[i] ← j – 1

Example due to Dr. Lori Pollock
Building a Control-Flow Graph

The first step in almost any optimization is building a CFG

// find all the leaders, assume first op & block are numbered zero
next ← 0 // block number
leader[next++] ← 0 // block’s 1st op

// find other leaders
for i ← 0 to n
  if op[i] is a jump
    then leader[next++] ← target(i)
  if op[i] is a branch then
    leader[next++] ← taken(i)
    leader[next++] ← not_taken(i)

// build all the blocks
for i ← 0 to next – 1
  j ← leader[i] + 1
  while j ≤ n and j ∉ leader
    j ← j + 1
  last[i] ← j – 1

EXAMPLE

0  a ← 4
1  t1 ← a * 4
2  L1: t2 ← t1 / c
3  if t2 < w then goto L2
4  m ← t1 * k
5  t3 ← m + i
6  L2: h ← i
7  m ← t3 – h
8  if t3 ≥ 0 then goto L3
9  goto L1
10 L3: halt

<table>
<thead>
<tr>
<th>LEADER</th>
<th>0</th>
<th>6</th>
<th>4</th>
<th>10</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAST</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Building a Control-Flow Graph

The first step in almost any optimization is building a CFG

// find all the leaders, assume first op & block are numbered zero
next ← 0       // block number
leader[next++] ← 0       // block’s 1st op

// find other leaders
for i ← 0 to n
    if op[i] is a jump
        then leader[next++] ← target(i)
    if op[i] is a branch then
        leader[next++] ← taken(i)
        leader[next++] ← not_taken(i)

// build all the blocks
for i ← 0 to next − 1
    j ← leader[i] + 1
    while j ≤ n and j ∉ leader
        j ← j + 1
    last[i] ← j − 1

EXAMPLE

0   a ← 4
1   t1 ← a * 4
2   L1: t2 ← t1 / c
3   if t2 < w then goto L2
4   m ← t1 * k
5   t3 ← m + i
6   L2: h ← i
7   m ← t3 − h
8   if t3 ≥ 0 then goto L3
9   goto L1
10  L3: halt

<table>
<thead>
<tr>
<th>LEADER</th>
<th>0</th>
<th>6</th>
<th>4</th>
<th>10</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAST</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>10</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>
Building a Control-Flow Graph

The first step in almost any optimization is building a CFG

// find all the leaders, assume first op & block are numbered zero
next ← 0          // block number
leader[next++] ← 0 // block’s 1st op

// find other leaders
for i ← 0 to n
  if op[i] is a jump
    then leader[next++] ← target(i)
  if op[i] is a branch then
    leader[next++] ← taken(i)
    leader[next++] ← not_taken(i)

// build all the blocks
for i ← 0 to next − 1
  j ← leader[i] + 1
  while j ≤ n and j ∉ leader
    j ← j + 1
  last[i] ← j − 1

EXAMPLE

L1: t2 ← t1 / c
if t2 < w then goto L2
m ← t1 * k
t3 ← m + i
L2: h ← i
m ← t3 − h
if t3 ≥ 0 then goto L3
goto L1
L3: halt

<table>
<thead>
<tr>
<th>LEADER</th>
<th>0</th>
<th>6</th>
<th>4</th>
<th>10</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAST</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>10</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>
Building a Control-Flow Graph

The first step in almost any optimization is building a CFG.

To build the control-flow graph, add code after assignments to leader:

// find all the leaders, assume first op // & block are numbered zero
next ← 0 // block number
leader[next++] ← 0 // block’s 1st op

// find other leaders
for i ← 0 to n
    if op[i] is a jump
        then leader[next++] ← target(i)
    if op[i] is a branch then
        leader[next++] ← taken(i)
        leader[next++] ← not_taken(i)

// build all the blocks
for i ← 0 to next − 1
    j ← leader[i] + 1
    while j ≤ n and j ∉ leader
        j ← j + 1
    last[i] ← j − 1

EXAMPLE

0  a ← 4
1  t1 ← a * 4
2  L1:  t2 ← t1 / c
3  if t2 < w then goto L2
4  m ← t1 * k
5  t3 ← m + i
6  L2:  h ← i
7  m ← t3 − h
8  if t3 ≥ 0 then goto L3
9  goto L1
10 L3:  halt
Local Value Numbering

An Old Idea

The key notion

(Balke 1967 or Ershov 1954)

- Assign an identifying number, $\text{VN}(e)$, to each expression $e$
  - $\text{VN}(x+y) = \text{VN}(j)$ iff $x+y$ and $j$ always have the same value
  - Use hashing over value numbers to make it efficient

- Use the value 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
  
  ($\text{build a DAG, § 8.3.1 in EaC1e}$)
Local Value Numbering

The Algorithm

For each operation $o$ in the block

1. Get value numbers for the operands from a 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 & use a “load immediate”

If hashing behaves, the algorithm runs in linear time
   ♦ If you don’t believe in hashing, try multi-set discrimination

Minor issues

• Commutative operator $\Rightarrow$ hash operands in each order or sort the operands by their VN$s$ before hashing  (either works, sorting is likely cheaper)
• Looks at operand’s value number, not its name

\[\text{EaC2e: digression on page 256 or reference [65]}\]
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 ↦ x + y</td>
<td>a³ ↦ x¹ + y²</td>
<td>a³ ↦ x¹ + y²</td>
</tr>
<tr>
<td>* b ↦ x + y</td>
<td>* b³ ↦ x¹ + y²</td>
<td>* b³ ↦ a³</td>
</tr>
<tr>
<td>a ↦ 17</td>
<td>a⁴ ↦ 17⁴</td>
<td>a⁴ ↦ 17⁴</td>
</tr>
<tr>
<td>* c ↦ x + y</td>
<td>* c³ ↦ x¹ + y²</td>
<td>* c³ ↦ a³ (oops!)</td>
</tr>
</tbody>
</table>

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

Options:
- Use c³ ↦ b³
- Save a³ in t³
- Rename around it

\[ a \leftarrow 17 \textbf{kills} \text{ the copy of value 3 in } a \]
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 17^4)</td>
<td>(a_1^4 \leftarrow 17^4)</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 works
Simple Extensions to Local Value Numbering

Constant folding

• Add a field to the table 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 the input’s VN
• Build a decision tree on operation

Identities:
(Click)
\(x \leftarrow y, x+0, x-0, x*1, x\div1, x-x, x*0, x\div x,\)
\(x\lor0, x \land 0xFF...FF, \max(x,\text{MAXINT}),\)
\(\min(x,\text{MININT}), \max(x,x), \min(y,y),\)
and so on ...

(over value numbers, not names)
But, Wait a Minute

If local value numbering renames everything, how does the program work after optimization?

• The flow of data across blocks is encoded into the names given to values
• While LVN is local, renaming is not

Different approaches to the problem of names

• Skip renaming and accept that LVN will miss some opportunities
• Differentiate “local” names from “global” names & rename local ones
  ♦ Requires analysis of entire procedure to discover difference between names
  ♦ Global name appears in two different basic blocks
• Skip renaming and complicate the algorithm for LVN
• Adopt a global approach to renaming before applying LVN
• Handle the problem with code shape

1 The analysis is simple, based on Briggs’ classification of names in semi-pruned SSA construction.
2 The algorithm could track all names associated with a value, updating that map on each assignment.
3 For example, the compiler could convert the code to an SSA name space.
**Code Shape**

The compiler can create a code shape that sidesteps this naming issue

- Each program variable gets a unique permanent name  
  “name” ≡ virtual register
- Evaluate each subexpression into a local, temporary name
- For each assignment, copy the **RHS** expression into the name associated with the **LHS** variable
  - Evaluations are the result of operations other than a register-to-register copy
  - Assignments are the result of a register-to-register copy
  - Connections across blocks all result from assignment
  - **LVN** leaves the copy operations alone

To make this easy:

- Assign virtual register numbers to variables, starting at low number
- Assign any temporary value a number higher than any variable
- Assignment becomes a copy from higher numbered register to lower one

The “code shape” solution may find fewer redundancies than either differentiating local & global names or building **SSA** form. (A program variable might be used in just 1 block.)  

See p. 248 in **EaC2e**
In discussing any optimization, we should look at three issues.

**Safety:** Does it change the results of the computation?
- Safety is proven with results of analysis
- Data-flow analysis or other special case analysis

**Profitability:** Is it expected to speed up execution?
- Many authors assume transformations are always profitable
- Use either heuristics or a careful algebra of costs

**Opportunity:** Can we efficiently locate places to apply it?
- Can we find find all the places where the transformation works?
- Do we need to update safety information afterward?
Safety

The first principle of code optimization:

The compiler must preserve the code’s “meaning”

When can the compiler transform the code?

• Original & transformed code must have the same final state
• Variables that are visible at exit
• Equality of result, not equality of method (ignore temporaries)

Formal notion

For two expressions, M and N, we say that M and N are observationally equivalent if and only if, in any context C where both M and N are closed (that is, have no free variables), evaluating C[M] and C[N] either produces identical results or neither terminates. 

Plotkin, 1975

⇒ Different translations with identical results are fine
Safety

In practice, compilers use a simpler notion of equivalence

*If, in the actual program context, the result of evaluating $e'$ cannot be distinguished from the result of evaluating $e$, the compiler can replace $e$ with $e'$.*

- This restatement ignores divergence
- If $e'$ is faster than $e$, the transformation is profitable

Equivalence and context

- Compiled code always executes in some context
- Optimization is the art of capitalizing on context
- Lack of context $\Rightarrow$ fully general (*i.e.*, slow) code

Some compilers employ a worse standard

- Correct behavior for “standard conforming” code
- Undefined behavior for other code
Safety

My favorite bad quote on safety

You, as a compiler writer, must decide if it’s worth the risk of doing this kind of optimization. It’s difficult for the compiler to distinguish between the safe and dangerous cases, here. For example, many C compilers perform risky optimizations because the compiler writer has assumed that a C programmer can understand the problems and take steps to remedy them at the source code level. It’s better to provide the maximum optimization, even if it’s dangerous, than to be conservative at the cost of less efficient code. A Pascal programmer may not have the same level of sophistication as a C programmer, however, so the better choice in this situation might be to avoid the risky optimization entirely or to require a special command-line switch to enable the optimization.


The point

• You must not violate the first principle
• Without the first principle, just compile a return and be done

This statement was gone by the 2nd Edition.
Safety & Local Value Numbering

Why is local value numbering safe?

• Hash table starts empty
• Expressions placed in table as processed
• If \(<\text{operator}, \text{VN}(o_1), \text{VN}(o_2)>\) is in the table, then
  ♦ It has already occurred at least once in the block
  ♦ Neither \(o_1\) nor \(o_2\) have been subsequently redefined
    → The mapping uses \(\text{VN}(o_1)\) and \(\text{VN}(o_2)\), not \(o_1\) and \(o_2\)
    → If one was redefined, it would have a new \(\text{VN}\)

If \(<\text{operator}, \text{VN}(o_1), \text{VN}(o_2)>\) has a \(\text{VN}\), the compiler can safely use it

• The algorithm incrementally constructs a proof that \(<\text{operator}, \text{VN}(o_1), \text{VN}(o_2)>\) is redundant
• The algorithm modifies the code, but does not invalidate the table

Critical property of a basic block: If any statement executes, they all execute, in a predetermined order
Profitability

The compiler should only transform the code when it helps!

- Eliminating one or more operations
- Replacing an operation with a cheaper one
- Moving an operation to a place where it will execute fewer times

Sometimes, we can prove profitability

- Fold a constant expression into an immediate operation

Sometimes, we must guess

- Eliminating a redundant operation in a loop

Sometimes, we cannot tell ...

- Inlining in a Fortran compiler

Compiler writers need to think explicitly about profitability ...

We should know when we cannot tell if some transformation is profitable!
Profitability & Local Value Numbering

When is local value numbering profitable?

• If reuse is cheaper than re-computation
  ♦ Does not cause a spill or a copy (hard to determine)
  ♦ In practice, assumed to be true

• Local constant folding is always profitable
  ♦ Re-computing uses a register, as does load immediate
  ♦ Immediate form of operation avoids even that cost

• Algebraic identities
  ♦ If it eliminates an operation, it is profitable
    \((x + 0)\)
  ♦ Profitability of simplification depends on target
    \((2x \Rightarrow x+x)\)
  ♦ Easy to factor target machine costs into the implementation
    \(\rightarrow do\text{'t apply it unless it is profitable!}\)
Opportunity

To perform an optimization, the compiler must locate all the places in the code where it can be applied

• Allows compiler to evaluate each possible application
• Leads to efficient application of the transformation
• Avoids additional search

Approaches

• Perform analysis to find opportunities
  ♦ **VERYBUSY** expressions & code hoisting
• Look at every operation
  ♦ Value numbering, loop invariant code motion
• Iterate over subset of the IR
  ♦ Operator strength reduction on **SSA** form
Opportunity & Local Value Numbering

How does local value numbering find opportunity?

• Linear scan over block, in execution order
• Constructs a model of program state
• At each operation, check for several opportunities

Summary

• It performs an exhaustive search of the opportunities
• This answer is not satisfying, but it is true
  ♦ Must limit cost of checking each operation
  ♦ For example, use a tree of algebraic identities by operator
• Hashing keeps cost down to $O(1)$ per operand + per operation