Local Optimization: Value Numbering

The “Desert Island” Optimization

Comp 412

Tutorial tonight, location TBA on Piazza.
Second exam will cover material from the midterm cutoff through Wednesday’s lecture.
The Optimizer

Typical Transformations

• Discover & propagate a constant value
• Move evaluation to a less-frequently executed place in the code
• Specialize code based on context
• Find & remove redundant code
• Remove useless or unreachable code
• Take advantage of a processor feature

Researchers have published hundreds of techniques, if not thousands.
Optimizers Work at Several Different Scopes

**Local optimization**
- Operates entirely within a single basic block
- Properties of block lead to strong optimizations

**Regional optimization**
- Operate on a region in the CFG that contains multiple blocks
- Loops, trees, paths, extended basic blocks

**Whole procedure optimization** *(intraprocedural or global)*
- Operate on entire CFG for a procedure
- Presence of cyclic paths forces analysis then transformation

**Whole program optimization** *(interprocedural or universal)*
- Operate on some or all of the call graph *(multiple procedures)*
- Must contend with call/return & parameter binding

---

**Yet another meaning for “scope”**

A basic block is a maximal length sequence of straight-line code.

A control-flow graph (CFG) has a node for each basic block and an edge for each branch or jump.

A call graph has a node for each procedure and an edge for each call.

---

COMP 412, Fall 2019
In scanning and parsing, “scope” refers to a region of the code that corresponds to a distinct name space.

→ Recall that most **ALLs** and **OOLs** follow *lexical scope* rules ...

---

In optimization “scope” refers to a region of the code that is subject to analysis and transformation.

- Notions are somewhat related
- Connection is not necessarily intuitive

Different scopes introduces different challenges & different opportunities

**Historically, optimization has been performed at several distinct scopes.**
**Today’s lecture looks at one local optimization.**
The Optimizer

Today: Local Value Numbering

- Discover & propagate a constant value
- Move evaluation to a less-frequently executed place in the code
- Specialize code based on context
- Find & remove redundant code
- Remove useless or unreachable code
- Take advantage of a processor feature

LVN accomplishes three goals, in one simple, inexpensive techniques
The Meta Issue

In optimization, the meta issue is much more straightforward.

- The optimizer takes in code from earlier in the compiler.
- It produces code for consumption by later phases of the compiler.

At compile time:

- The optimizer analyzes code to understand its runtime behavior
- The optimizer rewrites code to improve its runtime behavior

At runtime:

- The code runs. Full stop.

Some systems (JITs) try to optimize code while it is executing. That is beyond the scope of this lecture, and of chapters 8, 9, and 10. We may discuss JITs in the last week of class.
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 the expression $x+y$ is redundant, or available
• Rewriting the code to eliminate the redundant evaluation

One local technique for accomplishing both is called value numbering
Local Value Numbering

Performing redundancy elimination in the local context works well

• Within a block, compiler understands the execution order
  – Blocks that execute before or after the current block are uncertain territory
  – Any value created inside the block is certain

• Much of the available improvement can be caught locally
  – Redundancy in address expressions, constant folding
  – Algebraic simplification of expressions

What, then, is the role of larger scopes?

• Some opportunities need more context than a block
  – Code motion & placement are clearly non-local
  – Regional optimizations, such as improving a loop nest
  – Removing useless or unreachable code requires a larger scope

• Discovering and using knowledge about the “uncertain territory”
  – Serious opportunities exist, but compiler should get local ones first

Discovering constants at the whole procedure or whole program scope is profitable, but folding them is local.
Value Numbering

The Key Notion

• Assign an identifying number, \( VN(n) \), to each expression
  – \( VN(x+y) = VN(j) \ iff \ x+y \ and \ j \ always \ have \ the \ same \ value \)
  – Use hashing over the value numbers to make it efficient
• Use these numbers to improve the code

Improving the Code

• Replace redundant expressions
  – Same \( VN \Rightarrow \) refer to prior value rather than recompute
• Simplify algebraic identities
• Discover constant-valued expressions, fold & propagate them

This technique is designed for low-level, linear IRs, similar methods exist for trees (e.g., build a DAG, a directed acyclic graph)
Local Value Numbering

The Base Algorithm

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

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

If hashing behaves, the algorithm runs in linear time
- Hashing 3 small integers is a hard case
- If not, use multi-set discrimination\(^1\) or acyclic DFAs\(^2\)

Handling algebraic identities
- Case statement on operator type
- Handle special cases within each operator

And, of course, give the result a value number

Constant time per operation
use a proven hash function & test it

\(^1\) see p. 256 in EaC2e

\(^2\) see § 2.6.3 on p. 77 in EaC2e
Local Value Numbering

A Simple 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^3 \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 exprs 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)

<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>( \ast ) ( b_0 \leftarrow x_0 + y_0 )</td>
<td>( \ast ) ( b_0^3 \leftarrow x_0^1 + y_0^2 )</td>
<td>( \ast ) ( 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>( \ast ) ( c_0 \leftarrow x_0 + y_0 )</td>
<td>( \ast ) ( c_0^3 \leftarrow x_0^1 + y_0^2 )</td>
<td>( \ast ) ( 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
- Simple rewriting now works
Renaming and LVN

In the local labs, you learned about renaming

• Renaming fits rather naturally into the local allocator
  – Choose a “name space” that matches the creation & destruction of values

• The right name space simplifies the transformation on the code

This insight has profound consequences

• The compiler’s choice of name space can differ from the programmer’s
  – With SSA, this notion became a theme in optimization literature
  – It has led to simpler, more effective algorithms

• The compiler may choose different name spaces at different times
  – Name space for one situation may not be best for another
  – Translation between name spaces has both costs and benefits

• In LVN, arbitrary names add myriad complications to the algorithm
  – Lists to keep track of all names that hold VN 3, and code to maintain them, or introduction of temporary names, copies, & a copy-coalescing problem

SSA: static single-assignment form; see Chapter 9
Local Value Numbering

The Algorithm

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

Complexity & Speed Issues

- “Get value numbers” — search versus hash
- “Hash \( <\text{op}, \text{VN}(o_1), \text{VN}(o_2)> \)” — search versus hash
- Copy folding — set value number of result
- Commutative ops — hash twice versus sorting the operands

Fast execution of the compiler’s code is critical.
As Randy Scarborough (FORTRAN H ENHANCED) put it, “a compiler is one of the few applications that executes literally hundreds of millions of times.”
Modern JITs create a direct tie between compiler speed & application speed.
Simple Extensions to Value Numbering

**Constant Folding**

- Add a field to the hash 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 input VN
- Build a decision tree on operation
  - No obvious way to hash

**Identities (on VNs)**

- $x \leftarrow y$, $x + 0$, $x - 0$, $x \times 1$, $x \div 1$, $x - x$, $x \times 0$, $x \div x$, $x \lor 0$, $x \\& 0xFF...FF$, $\max(x, \text{MAXINT})$, $\min(x, \text{MININT})$, $\max(x, x)$, $\min(y, y)$, and so on ...

*See Figure 8.3 on page 424 of EaC2e*
Local Value Numbering (Recap)

The LVN Algorithm, with bells & whistles

for \( i \leftarrow 1 \) to \( n \)
   1. get the value numbers \( V_1 \) for \( L_i \) and \( V_2 \) for \( R_i \)
   2. if \( O_{p_i} \) commutes and \( V_1 > V_2 \) then
      swap \( V_1 \) and \( V_2 \)
   3. construct a hash key \( <V_1, O_{p_i}, V_2> \)
   4. if the hash key is already present in the table then
      replace operation \( i \) with a copy into \( T_i \), and mark \( T_i \) with the VN
   5. if \( V_1 \) and \( V_2 \) are both marked as constant then
      evaluate \( L_i \, O_{p_i} \, R_i \), assign it to \( T_i \), enter it, and mark \( T_i \) as a constant
   6. if \( L_i \, O_{p_i} \, R_i \) matches an identity then
      replace operation \( i \) with the simplified version or a copy assignment, and enter it
   7. Else enter \( T_i \) and the hash key into the table

“enter it” means to create new table entries for \( T_i \) & the hash key and assign them a new value number

Block is a sequence of \( n \) operations of the form

\[ T_i \leftarrow L_i \, O_{p_i} \, R_i \]
Another Brief Example

In Lab 2, your allocator might have encountered, or generated, code that looked something like this sample:

```
...  
add   r1, r2   => r3  
loadl 32768   => r4  
store r3       => r4  
...  
loadl 32768   => r17 
load  r17      => r18 
add   r18, r12 => r19 
...  
```

The uses of the value 32768 are clearly redundant. If r4 is still alive at the load, the compiler could avoid the loadl that puts 4 into r17.

LVN can rewrite this code to avoid the second loadl. Renaming would not.

Tracking the spill through RAM is more complicated (see Lab 3).

LVN can recognize that VN(r17) = VN(r18), even with more complex address expressions.

In the early 1950s, Ershov built LVN-like capability into an assembler. It would simplify this example.
Value Numbering

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

LVN found these redundant ops
LVN missed these opportunities (need stronger methods)

Regional Value Numbering
- Requires the compiler to use knowledge across blocks
- Imposes an order on traversal of the graph
- Slightly more complicated algorithm (see §8.5.1)