Local Optimization: Value Numbering

The “Desert Island” Optimization

Comp 412
<table>
<thead>
<tr>
<th>Sunday</th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
<th>Saturday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oct 21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Docs Available Lab Lecture</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Tutorial 4PM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Correct Schedule</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Improve Schedules</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Lab 3 Due Date</td>
<td>No Class (Walk)</td>
<td>Thanksgiving Break</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Late day</td>
<td>Not a late day</td>
<td>Not a late day</td>
<td>Not a late day</td>
</tr>
</tbody>
</table>
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
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
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

Assume a low-level, linear IR such as ILOC
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
Value Numbering

The Key Notion

• Assign an identifying number, VN(n), to each expression
  – VN(x+y) = VN(j) \textit{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
• Technique designed for low-level, linear IRs, similar methods exist for trees (e.g., build a DAG, a \textit{directed acyclic graph})
Local Value Numbering

The 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 \text{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 ← 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 ← 17</td>
<td>a⁴ ← 17</td>
</tr>
<tr>
<td>* c ← x + y</td>
<td>* c³ ← x¹ + y²</td>
<td>* c³ ← a³ (oops!)</td>
</tr>
</tbody>
</table>

**Two redundancies**
- Eliminate exprs with a *
- Coalesce results ?

**Options**
- Use c³ ← b³
- Save a³ in t³
- 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>* ( 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
- Simple rewriting now works
Digression on Renaming

In the local register allocation lab, we introduced you to renaming

- Renaming fits rather naturally into that particular algorithm
  - 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
  - This notion has become a theme of optimization in the last 20 years
  - 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
Local Value Numbering

The 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

Complexity & Speed Issues
- “Get value numbers” — search versus hash
- “Hash \( \langle \text{op}, \text{VN}(o_1), \text{VN}(o_2) \rangle \)” — 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 JIT environments make the tie between compiler speed & application speed even more critical.
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)_

\[
\begin{align*}
x &\leftarrow y, \ x + 0, \ x - 0, \ x \cdot 1, \ x \div 1, \ x \cdot x, \ x \cdot 0, \ x \div x, \\
x \land 0, \ x \land 0xFF...FF, \ \text{max}(x, \text{MAXINT}), \\
\text{min}(x, \text{MININT}), \ \text{max}(x, x), \ \text{min}(y, y), \text{ and so on ...}
\end{align*}
\]

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

The LVN Algorithm, with bells & whistles

```
for i ← 1 to n
1. get the value numbers V₁ and V₂ for Lᵢ and Rᵢ
2. if Opᵢ commutes and V₁ > V₂ then
   swap V₁ and V₂
3. construct a hash key <V₁, Opᵢ, V₂>
4. if the hash key is already present in the table then
   replace operation i with a copy into Tᵢ, and mark Tᵢ with the VN
5. if V₁ and V₂ are both marked as constant then
   evaluate Lᵢ Opᵢ Rᵢ, assign it to Tᵢ, enter it, and mark Tᵢ as a constant
6. if Lᵢ Opᵢ Rᵢ matches an identity then
   replace operation i with the simplified version or a copy assignment, and
   enter it
7. Else enter Tᵢ and the hash key into the table
```

Block is a sequence of n operations of the form

```
Tᵢ ← Lᵢ Opᵢ Rᵢ
```

Commutativity

Constant folding

Algebraic identities

“enter it” means to create new table entries for Tᵢ & the hash key and assign them a new value number
Another Brief Example

In Lab 1, your allocator might have encountered, or generated, code that looked something like this sample:

```
...  
add  r1, r2 => r3  
loadI 4 => r4  
store  r3 => r4  
...  
loadI 4 => r17  
load  r17 => r18  
add  r18, r12 => r19  
...  
```

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

LVN can rewrite this code to avoid the second loadI.

Tracking the spill through RAM is more complicated, as you will learned in Lab 3.

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

LVN found these redundant ops

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

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)