Machine Independent Code Optimizations

Useless Code and Redundant Expression Elimination
The goal of code optimization is to
- Discover program run-time behavior at compile time
- Use the information to improve generated code
  - Speed up runtime execution of compiled code
  - Reduce the size of compiled code

Correctness (safety)
- Optimizations must preserve the meaning of the input code

Profitability
- Optimizations must improve code quality
Applying Optimizations

- Most optimizations are separated into two phases
  - Program analysis: discover opportunity and prove safety
  - Program transformation: rewrite code to improve quality
- The input code may benefit from many optimizations
  - Every optimization acts as a filtering pass that translates one IR into another IR for further optimization
- Compilers
  - Select a set of optimizations to implement
  - Decide orders of applying implemented optimizations
    - The safety of optimizations depends on results of program analysis
    - Optimizations often interact with each other and need to be combined in specific ways
    - Some optimizations may need to be applied multiple times
      - E.g., dead code elimination, redundancy elimination, copy folding
  - Implement predetermined passes of optimizations
Scalar Compiler Optimizations

- **Machine independent optimizations**
  - Enable other transformations
    - Procedure inlining, cloning, loop unrolling
  - Eliminate redundancy
    - Redundant expression elimination
  - Eliminate useless and unreachable code
    - Dead code elimination
  - Specialization and strength reduction
    - Constant propagation, peephole optimization
  - Move operations to less-frequently executed places
    - Loop invariant code motion

- **Machine dependent (scheduling) transformations**
  - Take advantage of special hardware features
    - Instruction selection, prefetching
  - Manage or hide latency, introduce parallelism
    - Instruction scheduling, prefetching
  - Manage bounded machine resources
    - Register allocation
Scope Of Optimization

- **Local methods**
  - Applicable only to basic blocks

- **Superlocal methods**
  - Operate on extended basic blocks (EBB) 
    \[ B_1, B_2, B_3, \ldots, B_m \text{, where } B_i \text{ is the single predecessor of } B(i+1) \]

- **Regional methods**
  - Operate beyond EBBs, e.g. loops, conditionals

- **Global (intraprocedural) methods**
  - Operate on entire procedure (subroutine)

- **Whole-program (interprocedural) methods**
  - Operate on entire program
Loop Unrolling

- An enabling transformation to expose opportunities for other optimizations
  - Reduce the number of branches by a factor 4
  - Provide a bigger basic block (loop body) for local optimization
    - Better instruction scheduling and register allocation

Original loop

```
do i = 1 to n by 1
  a(i) = a(i) + b(i)
end
```

Unrolled by 4, n = 100

```
do i = 1 to 100 by 4
  a(i) = a(i) + b(i)
  a(i+1) = a(i+1) + b(i+1)
  a(i+2) = a(i+2) + b(i+2)
  a(i+3) = a(i+3) + b(i+3)
end```

Loop Unrolling --- arbitrary n

do i = 1 to n-3 by 4
  a(i) = a(i) + b(i)
  a(i+1) = a(i+1) + b(i+1)
  a(i+2) = a(i+2) + b(i+2)
  a(i+3) = a(i+3) + b(i+3)
end

Unrolled by 4, arbitrary n

i = 1
if (mod(n,2) > 0) then
  a(i) = a(i) + b(i)
  j=j+1
if (mod(n,4) > 1) then
  a(i) = a(i)+b(i)
  a(i+1)=a(i+1)+b(i+1)
i=i+2
end do i = i to n by 4
  a(i) = a(i) + b(i)
  a(i+1) = a(i+1) + b(i+1)
  a(i+2) = a(i+2) + b(i+2)
a(i+3) = a(i+3) + b(i+3)
end

Unrolled by 4, arbitrary n
Eliminating Redundant Expressions

Original code

\[
\begin{align*}
    m & := 2 \times y \times z \\
    n & := 3 \times y \times z \\
    o & := 2 \times y - z
\end{align*}
\]

Rewritten code

\[
\begin{align*}
    t0 & := 2 \times y \\
    m & := t0 \times z \\
    n & := 3 \times y \times z \\
    o & := t0 - z
\end{align*}
\]

- The second \(2\times y\) computation is redundant
- What about \(y \times z\)?
  - \(2\times y \times z \Rightarrow (2\times y) \times z \not= 2\times(y \times z)
  - \(3\times y \times z \Rightarrow (3\times y) \times z \not= 3\times(y \times z)
  - Change associativity may change evaluation result
    - For integer operations, optimization is sensitive to ordering of operands

- Typically applied only to integer expressions due to precision concerns
The Role Of Naming

(1) The expression `x+y' is redundant, but no longer available in `a' when being assigned to `c'

- Keep track of available variables for each value number
- Create new temporary variables for value numbers if necessary

(2) The expression 2*y is not redundant

- the two 2*y evaluation have different values

(3) Pointer Variables could point to anywhere

- If p points to y, then 2*y is no longer redundant
- All variables (memory locations) may be modified from modifying *p
- Pointer analysis ---reduce the set of variables associated with p
Eliminate Redundancy In Basic Blocks
Value numbering (1)

- Simulate the runtime evaluation of expressions
  - For every distinct runtime value, create a unique integer number as compile-time handle
- Use a hash table to map every expression e to a integer value number VN(e)
  - Represent the runtime value of expression
    \[ VN(e_1 \text{ op } e_2) = \text{unique_map}(\text{op}, VN(e_1), VN(e_2)) \]
- If an expression has a already-defined value number
  - It is redundantly evaluated and can be removed

```
a<3> := b<1> + c<2>;
b<5> := a<3> - d<4>;
c<6> := b<5> + c<2>;
d<5> := a<3> - d<4>;
a := b + c;
b := a - d;
c := b + c;
d := b;
```
Eliminate Redundancy In Basic Blocks
Value numbering (2)

for each expression e of the form result := opd1 op opd2

1. Find value numbers for opd1 and opd2
   if VN(opd1) or VN(opd2) is a constant or has a replacement variable
      replace opd1/opd2 with the value
2. Construct a hash key for expression e from op, VN(opd1) and VN(opd2)
3. if the hash key is already defined in hash table with a value number
   if (result is a temporary) then remove e
   else   replace e with a copy
          record the value number for result
else
   insert e into hash table with new value number
   record value number for result (set replacement variable of value number)

Extensions:
When valuating a hash key k for expression e
   if operation can be simplified, simplify the expression
   if op is commutative, sort operands by their value numbers
## Example: Value Numbering

### Code Snippet:

```
ADDR_LOADI @c → r9
INT_LOADA @i → r10
INT_LOADI 4 → r11
INT_MULTI r10 r11 → r12
INT_PLUS r9 r12 → r13
FLOAT_LOADI 0.0 → r14
FLOAT_STORE r14 → r13
```

### Table: Value-numbering

<table>
<thead>
<tr>
<th>OP</th>
<th>opd1</th>
<th>opd2</th>
<th>Value-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR_LOADI</td>
<td>@c</td>
<td></td>
<td>v1</td>
</tr>
<tr>
<td>ALOADA</td>
<td>@c</td>
<td></td>
<td>v2</td>
</tr>
<tr>
<td></td>
<td>r9</td>
<td></td>
<td>v2</td>
</tr>
<tr>
<td></td>
<td>@i</td>
<td></td>
<td>v3</td>
</tr>
<tr>
<td>ILOADA</td>
<td>@i</td>
<td></td>
<td>v4</td>
</tr>
<tr>
<td></td>
<td>r10</td>
<td></td>
<td>v4</td>
</tr>
<tr>
<td></td>
<td>r11</td>
<td></td>
<td>INT_4</td>
</tr>
<tr>
<td>......</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Value-numbering to Variable Assignment

```
ADDR_LOADI c → r9
INT_LOADA i → r10
INT_MULTI r10 4 → r12
INT_PLUS r9 r12 → r13
FLOAT_STOREI 0.0 → r13
```

<table>
<thead>
<tr>
<th>Value-number</th>
<th>variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td>r9</td>
</tr>
<tr>
<td>v3</td>
<td></td>
</tr>
<tr>
<td>v4</td>
<td>r10</td>
</tr>
<tr>
<td>v5</td>
<td>r12</td>
</tr>
<tr>
<td>v6</td>
<td>r13</td>
</tr>
</tbody>
</table>
Implementing Value Numbering

- Implementing value numbers
  - Two types of value numbers
    - Compile-time integer constants
    - Integers representing unknown runtime values
  - Use a tag (bit) to tell which type of value number

- Implementing hash table
  - Must uniquely map each expression to a value number
    - variable name → value number
    - (op, VN1, VN2) → value number
  - Evaluating hash key
    - int hash(const char* name);
    - int hash(int op, int vn1, int vn2);
  - Need to resolve hash conflicts if necessary

- Keeping track of variables for value numbers
  - Every runtime value number resides in one or more variables
  - Replace redundant evaluations with saved variables
Superlocal Value Numbering

Finding EBBs in control-flow graph
- AB, ACD, ACE, F, G
- Expressions can be in multiple EBBs

Need to restore state of hash table at each block boundary
- Record and restore
- Use scoped value table

Weakness: does not catch redundancy at node F

Algorithm

`ValueNumberEBB(b, tbl, VN)`

- `PushBlock(tbl, VN)`
- `ValueNumbering(b, tbl, VN)`
  - for each child `bi` of `b`
  - if `b` is the only parent of `bi`
    - `ValueNumberEBB(bi, tbl, VN)`
- `PopBlock(tbl, VN)`
Dominator-Based Value Numbering

The execution of C always precedes F
- Can we use value table of C for F?

Problem: variables in C may be redefined in D or E

Solution: rename variables so that each variable is defined once
- SSA: static single assignment

Similarly, can use table of A for optimizing G
Exercise: Value Numbering

```c
int A[100];
void fee(int x, int y) {
  int I = 0, j = i;
  int z = x + y, h =0;
  while (I < 100) {
    I = I + 1;
    if (y < x) j = z + y;
    h = x + y;
    A[I] = x + y;
  }
  return;
}
```
Global Redundancy Elimination

- Value numbering cannot handle cycles in CFG
  - Makes a single pass over all basic blocks in predetermined order

- Global redundancy elimination
  - Intra-procedural methods
    - Handles arbitrarily shaped CFG
  - Based on expression syntax, not value
    - The first and second \( y \times z \) considered identical expression despite different values
    - Different from value number approach
Global redundancy elimination

(1) Collect all expressions in the code, each expression given a unique temporary name
- Expressions in M:
  \( y \cdot z, y - z \)

(2) At each CFG point \( p \), determine the set of available expressions
- An expression \( e \) is available at \( p \) if every CFG path leading to \( p \) contains a definition of \( e \), and no operand of \( e \) is modified after the definition

(3) At each CFG point, replace redundant evaluation of available expressions with a copy of the temporary variables
Computing Available Expressions

- For each basic block \( n \), let
  - \( \text{DEExpr}(n) = \text{expressions evaluated by } n \text{ and available at exit of } n \)
  - \( \text{ExprKill}(n) = \text{expressions whose operands are modified by } n \) (killed by \( n \))

Goal: evaluate expressions available on entry to \( n \)
- \( \text{Avail}(n) = \bigcap (\text{DEExpr}(m) \cup (\text{Avail}(m) - \text{ExprKill}(m))) \)
  \( m \in \text{pred}(n) \)

for each basic block \( b_i \)
  - compute \( \text{DEExpr}(b_i) \) and \( \text{ExprKill}(b_i) \)
  - if \( b_i \) is entry \( \text{Avail}(b_i) = \emptyset \) else \( \text{Avail}(b_i) = \text{domain} \)
for (changed := true; changed; )
  changed = false
for each basic block \( b_i \)
  oldAvail = \( \text{Avail}(b_i) \)
  \( \text{Avail}(b_i) = \bigcap (\text{DEExpr}(m) \cup (\text{Avail}(m) - \text{ExprKill}(m))) \)
  \( m \in \text{pred}(b_i) \)
  if \( \text{Avail}(b_i) \neq \text{oldAvail} \) changed := true
Exercise:
Global Redundancy Elimination

```c
int A[100];
void fee(int x, int y)
{
  int I = 0, j = i;
  int z = x + y, h =0;
  while (I < 100) {
    I = I + 1;
    if (y < x) j = z + y;
    h = x + y;
    A[I] = x + y;
  }
  return;
}
```
Useless/Dead Code Elimination

- Eliminate instructions whose results are never used
  - (1) mark all critical instructions as useful
    - Instructions that return values, perform input/output, or modify externally visible storage
  - (2) Mark all instructions that affect already-marked instruction i
    - Instructions that define operands of i or control the execution of i

```c
void foo(int b, int c) {
    int a, d, e, f;
    a := b + c;
    d := b - c;
    e := b * c;
    f := b / c;
    return e;
}
```

Useless code:
- a := b + c;
- d := b - c;
- f := b / c;
Useless/Dead Code Elimination Algorithm

Main:

MarkPass()
SweepPass()

MarkPass()
WorkList := ∅
for each operation i
    if i is critical then
        mark i; WorkList ∪ = {i}
while WorkList ≠ ∅
    remove i from WorkList
    let i be x := y op z
    if def(y) is not marked then
        mark def(y); WorkList∪={def(y)}
    if def(z) is not marked then
        mark def(z); WorkList∪={def(z)}
    for each branch j that
        controls execution of i
        if j is not marked then
            mark j; WorkList ∪= {j}

SweepPass()
for each operation i
    if i is unmarked then
        if i is a branch then
            rewrite i with a jump
            to i’s nearest marked postdominator
        if i is not a jump then
            delete i

Compute def(var): data-flow analysis or SSA.
Compute control(i): reverse dominance frontier analysis
Useless Code Elimination

Example

```
a = 5;
n := a + b
if (n < 10) goto 1

p := c + d
r := c + d

1: q := a + b
   r := c + d
   if (q < r) goto 2

2: e := b + 18
   s := a + b
   u := e + f
   goto 3

3: x := e + f
   Print x;
   if (x < 1) goto 1

5: y := a + b
   z := r + d
   return z
```

```
a = 5;
n := a + b
if (n < 10) goto 1

p := c + d
r := c + d

1: q := a + b
   r := c + d
   if (q < r) goto 2

2: e := b + 18
   s := a + b
   u := e + f
   e := a + 17
   u := e + f
   goto 3

3: x := e + f
   Print x;
   if (x < 1) goto 1

5: y := a + b
   z := r + d
   return z
```
Eliminating useless control flow

- Optimizations may introduce superfluous control flow
  - Eg., SSA conversion that breaks CFG edges

1. Folding redundant branch
2. Removing an empty block
3. Combining blocks
4. Hoisting a branch
Exercise: Useless Code Elimination

```c
int A[100];
void fee(int x, int y)
{
    int i = 0, j = i;
    int z = x + y, h = 0;
    while (i < 100) {
        i = i + 1;
        if (y < x) j = z + y;
        h = x + y;
        A[i] = x + y;
    }
    return;
}
```
Lazy code motion

- Move partially redundant code to less-frequently executed regions
  - Eg., move loop invariant code outside of loops

\[
\begin{align*}
  & b := b + 1 \\
  & a := b \times c \\
  & a := b \times c \\
  & \text{Partially redundant}
\end{align*}
\]

\[
\begin{align*}
  & b := b + 1 \\
  & a := b \times c \\
  & a := b \times c \\
  & \text{Reducant}
\end{align*}
\]
Lazy code motion --- algorithm

- Compute available expressions at the entry and exit of each basic block $n$
  - Expressions that can be safely moved forward along edges to $n$
  - Forward data flow analysis
- Compute anticipatable expressions at the entry and exit of each basic block
  - Expressions that can be safely moved backward along CFG edges to $n$
  - Backward dataflow analysis
- Compute the placement of expressions
  - Each CFG edge is annotated as the earliest location for placing a set of expressions (to be inserted into the edge)
  - Some expressions may be moved to later nodes (to be removed)
- Compute insertion and deletion sets
  - Insert expressions to CFG edges and remove expressions from CFG nodes
Availability and anticipatability analysis

Availability analysis: for each basic block $n$, let

- $\pi_{DEExpr}(n)=$ expressions evaluated by $n$ and available at exit of $n$
- $\pi_{ExprKill}(n)=$ expressions whose operands are modified by $n$
- Expressions available on entry to $n$ and on exit from $n$

$\pi_{AvailIn}(n) = \bigcap_{m \in \text{preds}(n)} \pi_{AvailOut}(m)$
$\pi_{AvailOut}(m) = \pi_{DEExpr}(m) \cup (\pi_{AvailIn}(m) - \pi_{ExprKill}(m))$

Anticipatability analysis: for each basic block $n$, let

- $\pi_{UEExpr}(n)=$ expressions used in $n$ without redefinition to operands
- $\pi_{ExprKill}(n)=$ expressions whose operands are modified by $n$
- Expressions available on entry to $n$ and on exit from $n$

$\pi_{AntOut}(n) = \bigcap_{m \in \text{succ}(n)} \pi_{AntIn}(m)$
$\pi_{AntIn}(m) = \pi_{UEExpr}(m) \cup (\pi_{AntOut}(m) - \pi_{ExprKill}(m))$
Placement of expressions

Earliest placement
- For an edge \(<bi,bj>\) in the CFG, an expression \(e \in \text{Earliest}(bi,bj)\) iff the computation can legally move to \(<bi,bj>\) and cannot move to any earlier edge.

\[
\text{Earliest}(bi,bj) = \text{AntIn}(bj) - \text{AvailOut}(bi) - (\text{AntOut}(bi) - \text{ExprKill}(bi))
\]

Later placement
- Can the earliest placement of an expression be moved forward in CFG without changing expression result?

\[
\text{LaterIn}(bj) = \bigcap_{bi \in \text{pred}(bj)} \text{Later}(bi,bj)
\]

\[
\text{Later}(bi,bj) = \text{Earliest}(bi,bj) \cup (\text{LaterIn}(bi) - \text{UEExpr}(bi))
\]
Rewrite the code

Compute insert set
- At each edge \((bi, bj)\), the set of expressions to insert evaluation
  \[ \text{Insert}(bi, bj) = \text{Later}(bi, bj) - \text{LaterIn}(bj) \]
  - If \(bi\) has a single successor, insert at the end of \(bi\)
  - If \(bj\) has a single predecessor, insert at the entry of \(bj\)
  - Otherwise, split \((bi, bj)\) and insert a new block

Compute delete set
- At each basic block \(bi\), the set of expressions to delete from \(bi\)
  \[ \text{Delete}(bi) = \text{UEExpr}(bi) - \text{LaterIn}(bi) \]
  - If \(e \in \text{Delete}(bi)\), then the upward-exposed evaluation of \(e\) is redundant in \(bi\) after all the insertions have been made. Remove all such evaluations with a reference to results of earlier evaluation
Example for lazy code motion

B1: loadI 1   => r1
     i2i   r1   => r2
loadAI r0, @m => r3
     i2i   r3   => r4
     cmp_LT r2, r4 => r5
     cbr   r5   => B2, B3
B2: mult r17, r18 => r20
     add   r19, r20 => r21
     i2i   r21   => r8
     addI  r2, 1   => r6
     i2i   r6    => r2
     cmp_GT r2, r4 => r7
     cbr   r7    => B3, B2
B3: ......

Set of expressions: r1, r3, r5, r6, r7, r20, r21

CFG:
Summary
Machine independent optimizations

- Eliminate redundancy
  - redundant expression elimination

- Specialize computation
  - Constant propagation, peephole optimization

- Eliminate useless and unreachable code
  - Dead code elimination

- Move operations to less-frequently executed places
  - Loop invariant code motion

- Enable other transformations
  - Inlining, cloning, loop unrolling
Appendix: Available Expression Analysis: Compute local sets

for each basic block \( n: S_1; S_2; S_3; \ldots; S_k \)

\[
\begin{align*}
\text{VarKill} & := \emptyset \\
\text{DEExpr}(n) & := \emptyset \\
& \text{for } i = k \text{ to } 1 \\
& \quad \text{suppose } S_i \text{ is } "x := y \text{ op } z" \\
& \quad \text{if } y \notin \text{VarKill} \text{ and } z \notin \text{VarKill} \\
& \qquad \text{DEExpr}(n) = \text{DEExpr}(n) \cup \{y \text{ op } z\} \\
& \qquad \text{VarKill} = \text{VarKill} \cup \{x\} \\
\text{ExprKill}(n) & := \emptyset \\
& \text{for each expression } e \text{ in the procedure} \\
& \quad \text{for each variable } v \in e \\
& \qquad \text{if } v \in \text{VarKill} \text{ then} \\
& \qquad \quad \text{ExprKill}(n) := \text{ExprKill}(n) \cup \{e\}
\end{align*}
\]
Appendix: Example: applying GRE

\[
\begin{align*}
  m & := a + b \\
  n & := a + b \\
  p & := c + d \\
  r & := c + d \\
  q & := a + b \\
  r & := c + d \\
  e & := b + 18 \\
  s & := a + b \\
  a & := e + f \\
  e & := a + 17 \\
  t & := c + d \\
  d & := e + f \\
  v & := a + b \\
  w & := c + d \\
  a & := e + f \\
  y & := a + b \\
  z & := c + d
\end{align*}
\]