Code Optimization

REQUIREMENTS:
• Meaning must be preserved (correctness)
• Speedup must occur on average.
• Work done must be worth the effort.

OPPORTUNITIES:
• Programmer (algorithm, directives)
• Intermediate code
• Target code
Code Optimization

Scanner (lexical analysis) → tokens → Parser (syntax analysis) → Syntactic structure → Semantic Analysis (IC generator) → Code Optimizer

Symbol Table

Code Generator → Target language
Levels

- Window – peephole optimization
- Basic block
- Procedural – global (control flow graph)
- Program level – intraprocedural (program dependence graph)
Peephole Optimizations

• Constant Folding
  \[ x := 32 \quad \text{becomes} \quad x := 64 \]
  \[ x := x + 32 \]

• Unreachable Code
  \[ \text{goto L2} \]
  \[ x := x + 1 \leftarrow \text{unneeded} \]

• Flow of control optimizations
  \[ \text{goto L1} \quad \text{becomes} \quad \text{goto L2} \]
  ...
  \[ L1: \text{goto L2} \]
Peephole Optimizations

• Algebraic Simplification
  \[ x := x + 0 \leftarrow \text{unneeded} \]

• Dead code
  \[ x := 32 \leftarrow \text{where } x \text{ not used after statement} \]
  \[ y := x + y \rightarrow y := y + 32 \]

• Reduction in strength
  \[ x := x \times 2 \rightarrow x := x + x \]
Peephole Optimizations

• Local in nature
• Pattern driven
• Limited by the size of the window
Basic Block Level

- Common Subexpression elimination
- Constant Propagation
- Dead code elimination
- Plus many others such as copy propagation, value numbering, partial redundancy elimination, …
Simple example: \( a[i+1] = b[i+1] \)

- \( t_1 = i + 1 \)
- \( t_2 = b[t_1] \)
- \( t_3 = i + 1 \)
- \( a[t_3] = t_2 \)

Common expression can be eliminated
Now, suppose \( i \) is a constant:

\[
\begin{align*}
\cdot \quad i &= 4 \\
\cdot \quad t_1 &= i + 1 \\
\cdot \quad t_2 &= b[t_1] \\
\cdot \quad a[t_1] &= t_2
\end{align*}
\]

Final Code:

\[
\begin{align*}
\cdot \quad i &= 4 \\
\cdot \quad t_2 &= b[5] \\
\cdot \quad a[5] &= t_2
\end{align*}
\]
Control Flow Graph - CFG

CFG = < V, E, Entry >, where

V = vertices or nodes, representing an instruction or basic block (group of statements).
E = (V x V) edges, potential flow of control
Entry is an element of V, the unique program entry

Two sets used in algorithms:
• Succ(v) = \{x in V | exists e in E, e = v \rightarrow x\}
• Pred(v) = \{x in V | exists e in E, e = x \rightarrow v\}
Definitions

- **point** - any location between adjacent statements and before and after a basic block.

- A **path** in a CFG from point $p_1$ to $p_n$ is a sequence of points such that $\forall j, 1 \leq j < n$, either $p_i$ is the point immediately preceding a statement and $p_{i+1}$ is the point immediately following that statement in the same block, or $p_i$ is the end of some block and $p_{i+1}$ is the start of a successor block.
c = a + b
d = a * c
i = 1

f[i] = a + b
c = c * 2
if c > d

g = a * c
g = d * d

i = i + 1
if i > 10

points

path
Optimizations on CFG

• Must take control flow into account
  – Common Sub-expression Elimination
  – Constant Propagation
  – Dead Code Elimination
  – Partial redundancy Elimination
  – …

• Applying one optimization may create opportunities for other optimizations.
Redundant Expressions

An expression \( x \text{ op } y \) is redundant at a point \( p \) if it has already been computed at some point(s) and no intervening operations redefine \( x \) or \( y \).

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

\[
\begin{align*}
t_0 &= 2\times y \\
t_0 &= 2\times y \\
t_0 &= 2\times y \\
t_1 &= 3\times y \\
t_1 &= 3\times y \\
t_2 &= 2\times y \\
t_0 &= t_0 \times z \\
t_1 &= t_1 \times z \\
t_0 &= t_0 \times z \\
&= t_0 - z
\end{align*}
\]
Redundant Expressions

Definition site

Since \(a + b\) is available here, \(\rightarrow\) redundant!

Candidates:

- \(a + b\)
- \(a * c\)
- \(d * d\)
- \(c * 2\)
- \(i + 1\)
Redundant Expressions

\[
c = a + b \\
d = a * c \\
i = 1 \\
f[i] = a + b \\
c = c * 2 \\
\text{if } c > d \\
g = a * c \\
c = d * d \\
i = i + 1 \\
\text{if } i > 10 \\
\]

Candidates:
- a + b
- a * c
- d * d
- c * 2
- i + 1

Definition site
Kill site
Not available
- Not redundant
Redundant Expressions

• An expression $e$ is defined at some point $p$ in the CFG if its value is computed at $p$. (definition site)
• An expression $e$ is killed at point $p$ in the CFG if one or more of its operands is defined at $p$. (kill site)
• An expression is available at point $p$ in a CFG if every path leading to $p$ contains a prior definition of $e$ and $e$ is not killed between that definition and $p$. 
Removing Redundant Expressions

\[ t_1 = a + b \]
\[ c = t_1 \]
\[ d = a \times c \]
\[ i = 1 \]
\[ f[i] = t_1 \]
\[ c = c \times 2 \]
\[ \text{if } c > d \]

Candidates:

- \( a + b \)
- \( a \times c \)
- \( d \times d \)
- \( c \times 2 \)
- \( i + 1 \)
Constant Propagation

b = 5
c = 4*b
c > b
d = b + 2
e = a + b

b = 5
c = 20
c > 5
d = 7
e = a + 5

b = 5
c = 20
20 > 5
d = 7
e = a + 5
Constant Propagation

b = 5
\[20 > 5\]
c = 20
d = 7
e = a + 5

b = 5
c = 20
d = 7
e = a + 5
Copy Propagation

\[
\begin{align*}
b &= a \\
c &= 4b \\
c &> b \\
d &= b + 2 \\
e &= a + b
\end{align*}
\]
Simple Loop Optimizations: Code Motion

```plaintext
while (i <= limit - 2)
  L1:
    t1 = limit - 2
    if (i > t1) goto L2
    body of loop
    goto L1
  L2:

t := limit - 2
  while (i <= t)
    L1:
      t1 = limit - 2
      if (i > t1) goto L2
      body of loop
      goto L1
```
Simple Loop Optimizations: Strength Reduction

- Induction Variables control loop iterations

```
j = j - 1

\text{if } t5 > v
\begin{align*}
t4 &= t4 - 4 \\
t5 &= a[t4]
\end{align*}
```

```
j = j - 1

\text{if } t5 > v
\begin{align*}
t4 &= 4*j \\
t5 &= a[t4]
\end{align*}
```
Simple Loop Optimizations

• Loop transformations are often used to expose other optimization opportunities:
  – Normalization
  – Loop Interchange
  – Loop Fusion
  – Loop Reversal
  – …
Consider Matrix Multiplication

for i = 1 to n do
    for j = 1 to n do
        for k = 1 to n do
        end
    end
end
Memory Usage

• **For A:** Elements are accessed across rows, spatial locality is exploited for cache (assuming row major storage)
• **For B:** Elements are accessed along columns, unless cache can hold all of B, cache will have problems.
• **For C:** Single element computed per loop – use register to hold

\[
C[i][j] = A[i][k] + B[k][j]
\]
Matrix Multiplication Version 2

for i = 1 to n do
    for k = 1 to n do
        for j = 1 to n do
        end
    end
end

loop interchange
Memory Usage

- **For A:** Single element loaded for loop body
- **For B:** Elements are accessed along rows to exploit spatial locality.
- **For C:** Extra loading/storing, but across rows
Simple Loop Optimizations

• How to determine safety?
  – Does the new multiply give the same answer?
  – Can be reversed??
    for (I=1 to N) a[I] = a[I+1] – can this loop be safely reversed?
Data Dependencies

• Flow Dependencies - write/read
  x := 4;
y := x + 1

• Output Dependencies - write/write
  x := 4;
x := y + 1;

• Antidependencies - read/write
  y := x + 1;
x := 4;
\[
x := 4 \\
y := 6 \\
p := x + 2 \\
z := y + p \\
x := z \\
y := p
\]
Global Data Flow Analysis

Collecting information about the way data is used in a program.

- Takes control flow into account
- HL control constructs
  - Simpler – syntax driven
  - Useful for data flow analysis of source code
- General control constructs – arbitrary branching

Information needed for optimizations such as: constant propagation, common sub-expressions, partial redundancy elimination …
Dataflow Analysis: Iterative Techniques

• First, compute local (block level) information.

• Iterate until no changes

while change do
    change = false
    for each basic block
        apply equations updating IN and OUT
        if either IN or OUT changes, set change to true
    end
Live Variable Analysis

A variable $x$ is \textit{live} at a point $p$ if there is some path from $p$ where $x$ is used before it is defined.

Want to determine for some variable $x$ and point $p$ whether the value of $x$ \textit{could} be used along some path starting at $p$.

- Information flows backwards
- May – ‘along some path starting at $p$’
Global Live Variable Analysis

Want to determine for some variable $x$ and point $p$ whether the value of $x$ could be used along some path starting at $p$.

- **DEF[B]** - set of variables assigned values in B prior to any use of that variable
- **USE[B]** - set of variables used in B prior to any definition of that variable
- **OUT[B]** - variables live immediately after the block
  \[ \text{OUT}[B] = \bigcup \text{IN}[S] \text{ for all } S \text{ in } \text{succ}(B) \]
- **IN[B]** - variables live immediately before the block
  \[ \text{IN}[B] = \text{USE}[B] + (\text{OUT}[B] - \text{DEF}[B]) \]
d1: a = 1

d2: b = 2

d3: c = a + b

d4: d = c - a

B1

DEF = a, b

USE =

B2

DEF = c, d

USE = a, b

B3

DEF =

USE = b, d

B4

DEF = d

USE = a, b, e

B5

DEF = e

USE = a, b, c

B6

DEF = a

USE = b, d

B8: b = a + b

B9: e = c - 1

B10: a = b * d

B22: b = a - d
Global Live Variable Analysis

Want to determine for some variable $x$ and point $p$ whether the value of $x$ could be used along some path starting at $p$.

- **DEF[B]** - set of variables assigned values in B prior to any use of that variable
- **USE[B]** - set of variables used in B prior to any definition of that variable
- **OUT[B]** - variables live immediately after the block
  \[ \text{OUT}[B] = \bigcup \text{IN}[S] \text{ for all } S \text{ in } \text{succ}(B) \]
- **IN[B]** - variables live immediately before the block
  \[ \text{IN}[B] = \text{USE}[B] \cup (\text{OUT}[B] - \text{DEF}[B]) \]
<table>
<thead>
<tr>
<th>Block</th>
<th>DEF</th>
<th>USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>{a,b}</td>
<td>{}</td>
</tr>
<tr>
<td>B2</td>
<td>{c,d}</td>
<td>{a,b}</td>
</tr>
<tr>
<td>B3</td>
<td>{}</td>
<td>{b,d}</td>
</tr>
<tr>
<td>B4</td>
<td>{d}</td>
<td>{a,b,e}</td>
</tr>
<tr>
<td>B5</td>
<td>{e}</td>
<td>{a,b,c}</td>
</tr>
<tr>
<td>B6</td>
<td>{a}</td>
<td>{b,d}</td>
</tr>
</tbody>
</table>

OUT[B] = \bigcup \text{IN}[S] \text{ for all } S \text{ in } \text{succ}(B)

IN[B] = \text{USE}[B] + (\text{OUT}[B] - \text{DEF}[B])
Dataflow Analysis Problem #2: Reachability

• A definition of a variable $x$ is a statement that may assign a value to $x$.
• A definition may reach a program point $p$ if there exists some path from the point immediately following the definition to $p$ such that the assignment is not killed along that path.
• Concept: relationship between definitions and uses
What blocks do definitions $d_2$ and $d_4$ reach?

$\begin{align*}
  d_1 i &= m - 1 \\
  d_2 j &= n \\
  d_3 i &= i + 1 \\
  d_4 j &= j - 1
\end{align*}$
Reachability Analysis: Unstructured Input

1. Compute GEN and KILL at block—level
2. Compute IN[B] and OUT[B] for B
   \[ \text{IN}[B] = \bigcup \text{OUT}[P] \quad \text{where}\, P\, \text{is a predecessor of} \, B \]
   \[ \text{OUT}[B] = \text{GEN}[B] \cup (\text{IN}[B] - \text{KILL}[B]) \]
3. Repeat step 2 until there are no changes to OUT sets
Reachability Analysis: Step 1

For each block, compute local (block level) information = GEN/KILL sets

- GEN[B] = set of definitions generated by B
- KILL[B] = set of definitions that can not reach the end of B

This information does not take control flow between blocks into account.
Reasoning about Basic Blocks

Effect of single statement: \( a = b + c \)
- Uses variables \{b,c\}
- **Kills all definitions of \{a\}**
- **Generates new definition (i.e. assigns a value) of \{a\}**

Local Analysis:
- Analyze the effect of each instruction
- Compose these effects to derive information about the entire block
Example

\[
\begin{align*}
d1 & \ i = m - 1 \\
d2 & \ j = n \\
d3 & \ a = u1 \\
d4 & \ i = i + 1 \\
d5 & \ j = j - 1 \\
d6 & \ a = u2 \\
d7 & \ i = u2
\end{align*}
\]

- \text{B1:} \ Gen = 1,2,3, \ Kill = 4,5,6,7
- \text{B2:} \ Gen = 4,5, \ Kill = 1,2,7
- \text{B3:} \ Gen = 6, \ Kill = 3
- \text{B4:} \ Gen = 7, \ Kill = 1,4
Reachability Analysis: Step 2

Compute IN/OUT for each block in a forward direction. Start with $IN[B] = \emptyset$

- $IN[B] = \text{set of defns reaching the start of } B$
  - $= \cup \text{(out[P]) for all predecessor blocks in the CFG}$
- $OUT[B] = \text{set of defns reaching the end of } B$
  - $= GEN[B] \cup (IN[B] – KILL[B])$

Keep computing IN/OUT sets until a fixed point is reached.
Reaching Definitions Algorithm

• Input: Flow graph with GEN and KILL for each block
• Output: in[B] and out[B] for each block.

For each block B do out[B] = gen[B], (true if in[B] = emptyset)
change := true;
while change do begin
    change := false;
    for each block B do begin
        in[B] := U out[P], where P is a predecessor of B;
        oldout = out[B];
        if out[B] != oldout then change := true;
    end
end
IN[B] = ∪(out[P]) for all predecessor blocks in the CFG
\[
\begin{array}{|l|l|l|l|}
\hline
& \text{IN} & \text{OUT} & \text{IN} & \text{OUT} \\
\hline
\text{B1} & \emptyset & 1,2,3 & \emptyset & 1,2,3 \\
\hline
\text{B2} & \emptyset & 4,5 & \text{OUT}[1]+\text{OUT}[4] \\
& & & = 1,2,3,7 \\
& & 4,5 + (1,2,3,7) \\
& & - 1,2,7) \\
& & = 3,4,5 \\
\hline
\text{B3} & \emptyset & 6 & \text{OUT}[2] = 3,4,5 \\
& & & 6 + (3,4,5 - 3) \\
& & & = 4,5,6 \\
\hline
\text{B4} & \emptyset & 7 & \text{OUT}[2]+\text{OUT}[3] \\
& & & = 3,4,5,6 \\
& & & 7 + (3,4,5,6 - 1,4) = 3,5,6,7 \\
\hline
\end{array}
\]

\[\text{IN}[B] = \bigcup (\text{out}[P]) \text{ for all predecessor blocks in the CFG}\]

\[\text{OUT}[B] = \text{GEN}[B] + (\text{IN}[B] - \text{KILL}[B])\]
<table>
<thead>
<tr>
<th></th>
<th>IN</th>
<th>OUT</th>
<th>IN</th>
<th>OUT</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>∅</td>
<td>1,2,3</td>
<td>∅</td>
<td>1,2,3</td>
<td>∅</td>
<td>1,2,3</td>
</tr>
<tr>
<td>B2</td>
<td>∅</td>
<td>4,5</td>
<td>1,2,3,7</td>
<td>3,4,5</td>
<td>OUT[1] + OUT[4] = 1,2,3,5,6,7</td>
<td>4,5 + (1,2,3,5,6,7-1,2,7) = 3,4,5,6</td>
</tr>
<tr>
<td>B3</td>
<td>∅</td>
<td>6</td>
<td>3,4,5</td>
<td>4,5,6</td>
<td>OUT[2] = 3,4,5,6</td>
<td>6 + (3,4,5,6 – 3) = 4,5,6</td>
</tr>
<tr>
<td>B4</td>
<td>∅</td>
<td>7</td>
<td>3,4,5,6</td>
<td>3,5,6,7</td>
<td>OUT[2] + OUT[3] = 3,4,5,6</td>
<td>7+(3,4,5,6 – 1,4) = 3,5,6,7</td>
</tr>
</tbody>
</table>

IN[B] = ∪(out[P]) for all predecessor blocks in the CFG
Forward vs. Backward

Forward flow vs. Backward flow

Forward: Compute OUT for given IN, GEN, KILL
  – Information propagates from the predecessors of a vertex.
  – Examples: Reachability, available expressions, constant propagation

Backward: Compute IN for given OUT, GEN, KILL
  – Information propagates from the successors of a vertex.
  – Example: Live variable Analysis
Forward vs. Backward Equations

Forward vs. backward

– Forward:
  • $\text{IN}[B]$ - process $\text{OUT}[P]$ for all $P$ in predecessors($B$)
  • $\text{OUT}[B] = \text{local} \cup (\text{IN}[B] \setminus \text{local})$

– Backward:
  • $\text{OUT}[B]$ - process $\text{IN}[S]$ for all $S$ in successor($B$)
  • $\text{IN}[B] = \text{local} \cup (\text{OUT}[B] \setminus \text{local})$
May vs. Must

May vs. Must

Must – true on **all paths**
Ex: constant propagation – variable must provably hold appropriate constant on all paths in order to do a substitution

May – true on **some path**
Ex: *Live variable analysis* – a variable is live if it could be used on some path; *reachability* – a definition reaches a point if it can reach it on some path
May vs. Must Equations

• May vs. Must
  – May – $\text{IN}[B] = \bigcup\text{(out}[P]\text{)}$ for all $P$ in $\text{pred}(B)$
  – Must – $\text{IN}[B] = \bigcap\text{(out}[P]\text{)}$ for all $P$ in $\text{pred}(B)$
• **Reachability**
  
  – \( \text{IN}[B] = \cup (\text{out}[P]) \) for all \( P \) in \( \text{pred}(B) \)
  
  – \( \text{OUT}[B] = \text{GEN}[B] + (\text{IN}[B] - \text{KILL}[B]) \)

• **Live Variable Analysis**
  
  – \( \text{OUT}[B] = \cup (\text{IN}[S]) \) for all \( S \) in \( \text{succ}(B) \)
  
  – \( \text{IN}[B] = \text{USE}[B] \cup (\text{OUT}[B] - \text{DEF}[B]) \)

• **Constant Propagation**
  
  – \( \text{IN}[B] = \cap (\text{out}[P]) \) for all \( P \) in \( \text{pred}(B) \)
  
  – \( \text{OUT}[B] = \text{DEF\_CONST}[B] \cup (\text{IN}[B] - \text{KILL\_CONST}[B]) \)
Discussion

• Why does this work?
  – Finite set – can be represented as bit vectors
  – Theory of lattices

• Is this guaranteed to terminate?
  – Sets only grow and since finite in size …

• Can we find ways to reduce the number of iterations?
Choosing visit order for Dataflow Analysis

In forward flow analysis situations, if we visit the blocks in depth first order, we can reduce the number of iterations.

Suppose definition d follows block path 3 → 5 → 19 → 35 → 16 → 23 → 45 → 4 → 10 → 17 where the block numbering corresponds to the preorder depth-first numbering.

Then we can compute the reach of this definition in 3 iterations of our algorithm.