CMPT 379
Compilers

Anoop Sarkar
http://www.cs.sfu.ca/~anoop
Code Optimization

• There is no fully optimizing compiler $O$

• Let’s assume $O$ exists: it takes a program $P$ and produces output $\text{Opt}(P)$ which is the smallest possible

• Imagine a program $Q$ that produces no output and never terminates, then $\text{Opt}(Q)$ could be: $L1$: goto $L1$

• Then to check if a program $P$ never terminates on some inputs, check if $\text{Opt}(P(i))$ is equal to $\text{Opt}(Q) = \text{Solves the Halting Problem}$

• Full Employment Theorem for Compiler Writers, see Rice(1953)
Optimizations

• Non-Optimizations
• Correctness of optimizations
  – Optimizations must not change the meaning of the program
• Types of optimizations
  – Local optimizations
  – Global dataflow analysis for optimization
  – Static Single Assignment (SSA) Form
• Amdahl’s Law
Non-Optimizations

```c
enum { GOOD, BAD };
extern int test_condition();

void check()
{
    int rc;
    rc = test_condition();
    if (rc != GOOD) {
        exit(rc);
    }
}
```

Which version of check runs faster?
Types of Optimizations

• High-level optimizations
  – function inlining

• Machine-dependent optimizations
  – e.g., peephole optimizations, instruction scheduling

• Local optimizations or Transformations
  – within basic block
Types of Optimizations

• Global optimizations or Data flow Analysis
  – across basic blocks
  – within one procedure (intraprocedural)
  – whole program (interprocedural)
  – pointers (alias analysis)
Maintaining Correctness

• What does this program output?

3

Not:

$ decafcc byzero.decaf

Floating exception

int main() {
    int x;
    if (false) {
        x = 3/(3-3);
    } else {
        x = 3;
    }
    print_int( x);
}
Peephole Optimization

• Redundant instruction elimination
  – If two instructions perform that same function and are in the same basic block, remove one
  – Redundant loads and stores
    li $t0, 3
    li $t0, 4
  – Remove unreachable code
    li $t0, 3
    goto L2
    ... (all of this code until next label can be removed)
Peephole Optimization

• Flow control optimization
  goto L1
  L1: goto L2
• Algebraic simplification
• Reduction in strength
  – Use faster instructions whenever possible
• Use of Machine Idioms
• Filling delay slots
Constant folding & propagation

• Constant folding
  – compute expressions with known values at compile time

• Constant propagation
  – if constant assigned to variable, replace uses of variable with constant unless variable is reassigned
Constant folding & propagation

- Copy Propagation

```
a := d + e
b := d + e
c := d + e
t := d + e
```

```
t := d + e
a := t
b := t
c := t
```
Transformations

- Structure preserving transformations
- Common subexpression elimination

\[
a := b + c\\
b := a - d\\
c := b + c\\
d := a - d \ (\Rightarrow b)
\]
Transformations

• Dead-code elimination (combines copy propagation with removal of unreachable code)

```plaintext
if (debug) { f(); } /* debug := false (as a constant) */
if (false) { f(); } /* constant folding */
using deadcode elimination, code for f() is removed
x := t3
    x := t3
    t4 := x  becomes  t4 := t3
```
Transformations

• Renaming temporary variables
  
  \[ t_1 := b+c \] can be changed to \[ t_2 := b+c \]
  replace all instances of \[ t_1 \] with \[ t_2 \]

• Interchange of statements
  
  \[ t_1 := b+c \quad \text{and} \quad t_2 := x+y \]
  \[ t_2 := x+y \] can be converted to \[ t_1 := b+c \]
Transformations

• Algebraic transformations
  \[ d := a + 0 \quad (\Rightarrow a) \]
  \[ d := d \times 1 \quad (\Rightarrow \text{eliminate}) \]

• Reduction of strength
  \[ d := a \times^2 \quad (\Rightarrow a \times a) \]
int main() {
    extern int f(int);
    int i;
    int *a;
    for (i = 0;
        i < 10;
        i = i + 1)
    {
        a[i] = f(i);
    }
}
Control Flow Graph in TAC

main:
  i = 0
L0:
  t1 = 10
  t2 = i < t1
  ifFalse t2 Goto L1
  t3 = 4
  t4 = t3 * i
  t5 = a + t4
  param i
  t6 = call f, 1
  pop 4
  *(t5) = t6
  t7 = 1
  i = i + t7
  goto L0
L1:
  return

Entry  
i = 0  
definition/gen  
L0:  
t1 = 10  
t2 = i < t1  
ifFalse t2 goto L1  
reaches  
t3 = 4  
t4 = t3 * i  
t5 = a + t4  
param i  
t6 = call f, 1  
pop 4  
*(t5) = t6  
reaches  
t7 = 1  
i = i + t7  
goto L0  
kill  
Exit
SSA Form

• *def-use* chains keep track of where variables were defined and where they were used

• Consider the case where each variable has only one definition in the intermediate representation

• One static definition, accessed many times

• Static Single Assignment Form (SSA)
SSA Form

• SSA is useful because
  – Dataflow analysis and optimization is simpler when each variable has only one definition
  – If a variable has N uses and M definitions (which use N+M instructions) it takes N*M to represent def-use chains
  – Complexity is the same for SSA but in practice it is usually linear in number of definitions
  – SSA simplifies the register interference graph
SSA Form

• Original Program

  a := x + y
  b := a - 1
  a := y + b
  b := x * 4
  a := a + b

• SSA Form

  a1 := x + y
  b1 := a1 - 1
  a2 := y + b1
  b2 := x * 4
  a3 := a2 + b2

what about conditional branches?
SSA Form

1: \( b := M[x] \)
   \( a := 0 \)

2: if \( b < 4 \)

3: \( a := b \)

4: \( c := a+b \)

1: \( b1 := M[x1] \)
   \( a1 := 0 \)

2: if \( b1 < 4 \)

3: \( a2 := b1 \)

4: \( a3 := \phi(a2, a1) \)
   \( c1 := a3 + b1 \)
Edge-split SSA Form

1: \( b := M[x] \)  
   \( a := 0 \)

2: if \( b < 4 \)

3: \( a := b \)

4: \( c := a + b \)

Unique Successor & Unique Predecessor

1: \( b1 := M[x1] \)  
   \( a1 := 0 \)

2: if \( b1 < 4 \)

3: \( a2 := b1 \)

4: \( a3 := \phi (a2, a1) \)  
   \( c1 := a3 + b1 \)

5:
SSA Form

• Conversion from a Control Flow Graph (created from TAC) into SSA Form is not trivial

• SSA creation algorithms:
  – Original algorithm by Cytron et al. 1986
  – Lengauer-Tarjan algorithm (see the Tiger book by Andrew W. Appel for more details)
  – Harel algorithm
Conversion to SSA Form

• Simple idea: add a \( \phi \) function for every variable at a join point
• A join point is any node in the control-flow graph with more than one predecessor
• But: this is wasteful and unnecessary.
Conversion to SSA

1: a := 0

2: b := a + 1
   c := c + b
   a := b * 2
   if a < N
   3: return c

b1 is never used, stmt can be deleted

1: a1 := 0

2: a3 := φ (a2, a1)
   b1 := φ (b0, b2)
   c2 := φ (c0, c1)
   b2 := a3 + 1
   c1 := c2 + b2
   a2 := b2 * 2
   if a2 < N
   3: return c1
Conversion to SSA

1: \( a := 0 \)

2: \( b := a + 1 \)
   \( c := c + b \)
   \( a := b \times 2 \)
   if \( a < N \)
   return \( c \)

b2 changes in each loop. SSA is not functional programming!
Dominance Relation

- $X$ dominates $Y$ if every path from the start node to $Y$ goes through $X$
- $D(X)$ is the set of nodes that $X$ dominates
- $X$ strictly dominates $Y$ if $X$ dominates $Y$ and $X \neq Y$
Dominance Relation

D(5)={6,7,8}

5 strictly dominates 6, 7, 8
Dominance Relation

D(5)={6,7,8}

5 strictly dominates 6, 7, 8
Dominance Property of SSA

• Essential property of SSA form is the definition of a variable must *dominate* use of the variable:
  – If $X$ is used in a $\varphi$ function in block $n$, then definition of $X$ dominates every predecessor of $n$
  – If $X$ is used in a non-$\varphi$ statement in block $n$, then the definition of $X$ dominates $n$. 
Dominance Frontier

• X strictly dominates Y if X dominates Y and X ≠ Y

• Dominance Frontier (DF) of node X is the set of all nodes Y such that:
  – X dominates a predecessor of Y, AND
  – X does not strictly dominate Y
Dominance Frontier

\[ D(5) = \{6, 7, 8\} \]

\[ S(6) = \{4, 8\} \]

\[ S(7) = \{8, 12\} \]

\[ S(8) = \{5, 13\} \]

\[ DF(5) = \{4, 12, 5, 13\} \]
Dominance Frontier

• Algorithm to compute DF(X):
  – Local(X) := set of successors of X who do not immediately dominate X
  – Up(X) := set of nodes in DF(X) that are not dominated by X’s immediate dominator.
  – DF(X) := Union of Local(X) & ( Union of Up(K) for all K that are children of X )
Dominance Frontier

• ComputeDF(X):
  
  $S := \{\} \quad // \text{empty set}$

  For each node Y in Successor(X):
    
    If Y is not immediately dominating X:
      
      $S := S + \{Y\} \quad // \text{this is Local}(X), + \text{means union}$

  For each child K of X in D(X): // X dominates K
    
    For each element Y in ComputeDF(K):
      
      If X does not dominate Y,
        
        $S := S + \{Y\} \quad // \text{this is Up}(X)$

    $DF(X) = S$
Dominance Frontier

• Dominance Frontier Criterion
  – If node X contains definition of some variable $a$, then any node Y in the DF(X) needs a $\phi$ function for $a$.

• Iterated Dominance Frontier
  – Since a $\phi$ function is itself a definition of a new variable, we must iterate the DF criterion until no nodes in the CFG need a $\phi$ function.
Placing $\phi$ Functions

1: $V := _; W := _$

2:  

3: $V := _$

4:  

5: $W := _$

6:  

7:  

DF(3) = \{7\}

Empty boxes indicate uses of variables $V, W$
Placing $\phi$ Functions

1: V:=_; W:=_

2:

3: V:=_

4:

5: W:=_

6:

7: V:= $\phi$(V,V)

DF(3)={7}

DF(5)={6}
Placing $\phi$ Functions

1: $V:=\_;$ $W:=\_$$\quad$ $DF(3)=\{7\}$

2:

3: $V:=\_$$\quad$ $DF(5)=\{6\}$

4:

5: $W:=\_$$\quad$ 6: $W:=\phi(W,W)$

7: $V:=\phi(V,V)$
Placing $\phi$ Functions

1: $V:=\_; W:=\_$

2: 

3: $V:=\_$

4: 

5: $W:=\_$

6: $W:= \phi(W,W)$

7: $V:= \phi(V,V); W:= \phi(W,W)$

DF(6)={7}
Rename Variables

1: V1:=_; W1:=_
2: 
3: V2:=_
4: 
5: W2:=_
6: W3:= φ(W1,W2)
7: V3:= φ(V1,V2);
   W4:= φ(W1,W3)

DF(6)={7}
Converting to SSA Form

Program

i:=1
j:=1
k:=0
while k<100:
  if j < 20:
    j:=i
    k:=k+1
  else:
    j:=k
    k:=k+1
return j

Control Flow Graph
Converting to SSA Form

Control Flow Graph

1: i := 1  j := 1
   k := 0
2: if k < 100
3: if j < 20
4: return j
5: j := i
   k := k+1
6: j := k
   k := k+1
7:

Dominance Relations

- D(1) = {2,3,4,5,6,7}
- D(2) = {3,4,5,6,7}
- D(3) = {5,6,7}
- D(4) = {}  
- D(5) = {}
- D(6) = {}  
- D(7) = {}
Converting to SSA

1: i := 1  j := 1  k := 0

2: if k < 100

3: if j < 20

4: return j

5: j := i
  k := k+1

6: j := k
  k := k+1

7: 

Control Flow Graph

Dominator Tree

Dominance Relations

• D(1) = \{2,3,4,5,6,7\}
• D(2) = \{3,4,5,6,7\}
• D(3) = \{5,6,7\}
• D(4) = \{\}
• D(5) = \{\}
• D(6) = \{\}
• D(7) = \{\}
Converting to SSA

Control Flow Graph

1: i := 1  j := 1
   k := 0

2: if k < 100

3: if j < 20

4: return j

5: j := i
   k := k+1

6: j := k
   k := k+1

7:

Dominance Relations

- D(1) = {2,3,4,5,6,7}
- D(2) = {3,4,5,6,7}
- D(3) = {5,6,7}
- D(4) = {}
- D(5) = {}
- D(6) = {}
- D(7) = {}

Dominance Frontier

- DF(1) = {}
- DF(2) = {2}
- DF(3) = {2}
- DF(4) = {}
- DF(5) = {7}
- DF(6) = {7}
- DF(7) = {2}
Converting to SSA Form

1: i := 1  j := 1  k := 0

2: if k2 < 100

3: if j < 20

4: return j

5: j := i  k := k+1

6: j := k  k := k+1

7:

Variable j in 5
DF(5) = { 7 }
Converting to SSA Form

1: i := 1  j := 1  k := 0

2:  
   if k2 < 100

3:  if j < 20

4:  return j

5:  j := i  k := k+1

6:  j := k  k := k+1

7:  j := φ(j, j)

Variable j in 5  DF(5) = { 7 }

Variable j in 7  DF(7) = { 2 }
Converting to SSA Form

1: i := 1  j := 1  k := 0

2: j := \phi(j, j)
   if k2 < 100

3: if j < 20

4: return j

5: j := i  k := k+1

6: j := k  k := k+1

7: j := \phi(j, j)
Converting to SSA Form

1: i := 1  j := 1  
k := 0

2: j := \phi(j, j)  
k := \phi(k, k)  
if k2 < 100

3: if j < 20

4: return j

5: j := i  
k := k+1

6: j := k  
k := k+1

7: j := \phi(j, j)  
k := \phi(k, k)

Variable k in 5  
DF(5) = { 7 }

Variable k in 7  
DF(7) = { 2 }

Variable k in 6  
DF(6) = { 7 }
Converting to SSA Form

1: i1 := 1  j1 := 1  k1 := 0

2: j2 := \phi(j4, j1)  
k2 := \phi(k4, k1)  
if k2 < 100

3: if j2 < 20

4: return j2

5: j3 := i1  
k3 := k2+1

6: j5 := k2  
k5 := k2+1

7: j4 := \phi(j3, j5)  
k4 := \phi(k3, k5)
Optimizations using SSA

• SSA form contains statements, basic blocks and variables

• Dead-code elimination
  – if there is a variable \( v \) with no uses and def of \( v \) has no side-effects, delete statement defining \( v \)
  – if \( z := \phi(x, y) \) then eliminate this stmt if no defs for \( x,y \)
Optimizations using SSA

- Constant Propagation
  - if $v := c$ for some constant $c$ then replace $v$ with $c$ for all uses of $v$
  - $v := \phi (c_1, c_2, ..., c_n)$ where all $c_i$ are equal to $c$ can be replaced by $v := c$
Optimizations using SSA

• Conditional Constant Propagation
  – In previous flow graph, is \( j \) always equal to 1?
  – If \( j = 1 \) always, then block 6 will never execute and so \( j := i \) and \( j := 1 \) always
  – If \( j > 20 \) then block 6 will execute, and \( j := k \) will be executed so that eventually \( j > 20 \)
  – Which will happen? Using SSA we can find the answer.
Optimizations using SSA

1: \( i_1 := 1 \) \( j_1 := 1 \)
   \( k_1 := 0 \)

2: \( j_2 := \phi(j_4, j_1) \)
   \( k_2 := \phi(k_4, k_1) \)
   if \( k_2 < 100 \)

3: if \( j_2 < 20 \)

4: return \( j_2 \)

5: \( j_3 := i_1 \)
   \( k_3 := k_2 + 1 \)

6: \( j_5 := k_2 \)
   \( k_5 := k_2 + 1 \)

7: \( j_4 := \phi(j_3, j_5) \)
   \( k_4 := \phi(k_3, k_5) \)
Optimizations using SSA

1: \(i_1 := 1\) \(j_1 := 1\) \(k_1 := 0\)

2: \(j_2 := \phi(j_4, 1)\)
   \(k_2 := \phi(k_4, 0)\)
   if \(k_2 < 100\)

3: if \(j_2 < 20\)

4: return \(j_2\)

5: \(j_3 := 1\)
   \(k_3 := k_2 + 1\)

6: \(j_5 := k_2\)
   \(k_5 := k_2 + 1\)

7: \(j_4 := \phi(j_3, k_2)\)
   \(k_4 := \phi(k_3, k_5)\)
Optimizations using SSA

1: i1 := 1  j1 := 1  k1 := 0
2: j2 := φ(j4, 1)  k2 := φ(k4, 0)  if k2 < 100
3: if j2 < 20
4: return j2
5: j3 := 1  k3 := k2+1
6: k5 := k2+1
7: j4 := φ(j3, k2)  k4 := φ(k3, k5)
Optimizations using SSA

1: i1 := 1  j1 := 1  
k1 := 0

2:  
j2 := φ(j4, 1)  
k2 := φ(k4, 0)  
if k2 < 100

3: if j2 < 20

4: return j2

5: j3 := 1  
k3 := k2+1

6:  
k5 := k2+1

7:  
j4 := φ(1, k2)  
k4 := φ(k3,k5)
Optimizations using SSA

1: \texttt{i1 := 1 \ j1 := 1}
\texttt{k1 := 0}

2: \texttt{j2 := \phi(j4, 1)}
\texttt{k2 := \phi(k4, 0)}
\texttt{if k2 < 100}

3: \texttt{if j2 < 20}

4: \texttt{return j2}

5: \texttt{j3 := 1}
\texttt{k3 := k2+1}

7: \texttt{j4 := \phi(1)}
\texttt{k4 := \phi(k3)}
Optimizations using SSA

1: i1 := 1  j1 := 1  k1 := 0

2: j2 := φ(1, 1)  k2 := φ(k4, 0)
   if k2 < 100

3: if j2 < 20

4: return j2

5: j3 := 1  k3 := k2+1

7: k4 := φ(k3)
Optimizations using SSA

1: i1 := 1  j1 := 1  k1 := 0

2:
   k2 := φ(k4, 0)
   if k2 < 100

3: if 1 < 20

4: return 1

5:
   k3 := k2+1

6:
   k4 := φ(k3)
Optimizations using SSA

1: 

2: \( k_2 := \phi(k_4, 0) \) if \( k_2 < 100 \)

3: 

4: return 1

5: \( k_3 := k_2 + 1 \)

7: \( k_4 := \phi(k_3) \)
Optimizations using SSA

1: 

2: \( k_2 := \phi(k_3, 0) \) if \( k_2 < 100 \)

5: \( k_3 := k_2 + 1 \)

4: return 1
Optimizations using SSA

- Arrays, Pointers and Memory
  - For more complex programs, we need *dependencies*: how does statement B depend on statement A?
  - **Read after write**: A defines variable v, then B uses v
  - **Write after write**: A defines v, then B defines v
  - **Write after read**: A uses v, then B defines v
  - **Control**: A controls whether B executes
Optimizations using SSA

• Memory dependence
  \[ M[i] := 4 \]
  \[ x := M[j] \]
  \[ M[k] := j \]

• We cannot tell if \( i, j, k \) are all the same value which makes any optimization difficult

• Similar problems with Control dependence

• SSA does not offer an easy solution to these problems
More on Optimization

• Advanced Compiler Design and Implementation by Steven S. Muchnick

• Control Flow Analysis
• Data Flow Analysis
• Dependence Analysis
• Alias Analysis
• Early Optimizations
• Redundancy Elimination

• Loop Optimizations
• Procedure Optimizations
• Code Scheduling (pipelining)
• Low-level Optimizations
• Interprocedural Analysis
• Memory Hierarchy
Amdahl’s Law

- Speedup_{total} = ((1 - Time_{Fractionoptimized}) + Time_{Fractionoptimized}/Speedup_{optimized})^{-1}

- Optimize the common case, 90/10 rule
- Requires quantitative approach
  - Profiling + Benchmarking
- Problem: Compiler writer doesn’t know the application beforehand
Summary

- Optimizations can improve speed, while maintaining correctness
- Various early optimization steps
- Static Single-Assignment Form (SSA)
- Optimization using SSA Form