Concepts Introduced in Chapter 9

- introduction to compiler optimizations
- basic blocks and control flow graphs
- local optimizations
- global optimizations
Compiler Optimizations

• Compiler optimization is a misnomer.

• A code-improving transformation consists of a sequence of changes that preserves the semantic behavior (i.e. are safe).

• A code-improving transformation attempts to make the program
  – run faster
  – take up less space
  – consume less power

• An optimization phase consists of a sequence of code-improving transformations of the same type.
Places for Potential Improvement

User can:
- profile program
- change algorithm
- transform loops

Compiler can:
- improve loops
- procedure calls
- address calculations
- use registers
- select instructions
- do peephole transformations
Basic Blocks

• Basic block - a sequence of consecutive statements with exactly 1 entry and 1 exit

• leaders are instructions that start a new basic block
  – the first three-address instruction in the intermediate code is a leader
  – any instruction that is the target of a conditional or unconditional jump is a leader
  – any instruction that immediately follows a conditional or unconditional jump is a leader

followed by Fig. 8.7, 8.9
Control Flow

- Control flow graph - a directed graph where the nodes are basic blocks and block $B \rightarrow block \ C$ iff $C$ can be executed immediately after $B$
  - there is a jump from the end of $B$ to beginning of $C$
  - $C$ follows $B$ in program order
- $B$ is a *predecessor* of $C$, and $C$ is a *successor* of $B$
- Local optimizations - performed only within a basic block
- Global optimizations - performed across basic blocks
Example Control Flow Graph

```
sum<-0
j<-0
goto B3
```

```
t1<-j<<2
t2<-addr(a)
t3<-t2[t1]
sum<-sum+t3
```

```
if j<n goto B2
```

```
B1
B2
B3
```
Organization of the Code Optimizer

- **Front end**
- **Code optimizer**
- **Code generator**

- **Control-flow analysis**
- **Data-flow analysis**
- **Code transformations**
Types of Compiler Optimizations

- Function call
- Loop
- Memory access
- Control flow
- Data flow
- Machine specific
Function Call Optimizations

- Procedure integration or inlining
- Procedure specialization or cloning
- Tail recursion elimination
- Function memoization
Loop Optimizations

- Invariant code motion
- Strength reduction
- Induction variable elimination
- Unrolling
- Collapsing
- Fusion
- Software pipelining
Instruction Selection

- Accomplished by combining RTLs.
- Data dependences (links) are detected between RTLs.
- Pairs or triples of RTLs are symbolically merged.
- Legality is checked via a machine description.
Combining a Pair of RTLs

26 \[ r[1] = r[30]+i; \]
27 \{26\} \[ r[2] = M[r[1]]; \quad r[1]: \]

⇒


or

\[ r[2] = M[r[30]+i]; \quad r[1]: \]
Combining Three RTLs

31  \hspace{1cm} r[2] = M[r[3]]; \\
32 \{31\}  \hspace{1cm} r[2] = r[2]+1; \\
33 \{32\}  \hspace{1cm} M[r[3]] = r[2]; \hspace{1cm} r[2]: \\
   \Rightarrow \\
   M[r[3]] = M[r[3]]+1; \hspace{0.5cm} r[2] = M[r[3]]+1; \hspace{0.5cm} r[2]: \\
or \\
   M[r[3]] = M[r[3]]+1; \hspace{0.5cm} r[2]:
Cascading Instruction Selection

Actual example on PDP-11 (2 address machine)

38 \[ r[36]=r[5]; \]
39 \{38\} \[ r[36]=r[36]+i; \]
40 \[ r[37]=r[5]; \]
41 \{40\} \[ r[37]=r[37]+i; \]
42 \{41\} \[ r[40]=M[r[37]]; \]
43 \[ r[41]=1; \]
44 \{42\} \[ r[42]=r[40]; \]
45 \{43,44\} \[ r[42]=r[42]+r[41]; \]
46 \{45,39\} \[ M[r[36]]=r[42]; \]
Cascading Instruction Selection (cont.)

38 \[ r[36] = r[5]; \]
39 \{38\} \[ r[36] = r[36] + i; \]
40 \[ r[37] = r[5]; \]
42 \{40\} \[ r[40] = M[r[37] + i]; \] \[ r[37]: \]
43 \[ r[41] = 1; \]
44 \{42\} \[ r[42] = r[40]; \] \[ r[40]: \]
45 \{43,44\} \[ r[42] = r[42] + r[41]; \] \[ r[41]: \]
46 \{45,39\} \[ M[r[36]] = r[42]; \] \[ r[42]: r[36]: \]
Cascading Instruction Selection (cont.)

38 \hspace{1em} r[36]=r[5];
39 \hspace{1em} r[36]=r[36]+i;

42 \hspace{1em} r[40]=M[r[5]+i]];
43 \hspace{1em} r[41]=1;
44 \hspace{1em} r[42]=r[40]; \hspace{1em} r[40]:
45 \hspace{1em} r[42]=r[42]+r[41]; \hspace{1em} r[41]:
46 \hspace{1em} M[r[36]]=r[42]; \hspace{1em} r[42]:r[36]:
Cascading Instruction Selection (cont.)

38 \[ r[36]=r[5]; \]
39 {38} \[ r[36]=r[36]+i; \]

43 \[ r[41]=1; \]
44 \[ r[42]=M[r[5]+i]; \]
45 {43,44} \[ r[42]=r[42]+r[41]; \quad r[41]: \]
46 {45,39} \[ M[r[36]]=r[42]; \quad r[42]:r[36]: \]
Cascading Instruction Selection (cont.)

38    r[36]=r[5];
39 {38} r[36]=r[36]+i;

45 {44} r[42]=r[42]+1;
46 {45,39} M[r[36]]=r[42];        r[42]:r[36]:
Cascading Instruction Selection (cont.)

38 \hspace{1cm} r[36]=r[5];

44 \hspace{1cm} r[42]=M[r[5]+i]];

45 \hspace{1cm} r[42]=r[42]+1;

46 \hspace{1cm} M[r[36]+i]=r[42]; \hspace{1cm} r[42]:r[36]:
Cascading Instruction Selection (cont.)

45 \{44\} \( r[42] = r[42] + 1 \);
Cascading Instruction Selection (cont.)

Example Sequence of Optimizations

for (sum=0, j = 0; j < n; j++)
  sum = sum + a[j];

⇒ after instruction selection

  M[r[13] + sum] = 0;
  M[r[13] + j] = 0;
  PC = L18;
L19
  r[0] = M[r[13] + j] <<2;
L18
  PC = IC < 0 →L19;
Example Sequence of Optimizations (cont.)

⇒ after register allocation

\[
\begin{align*}
    r[2] &= 0; \\
    r[1] &= 0; \\
    PC &= L18;
\end{align*}
\]

L19

\[
\begin{align*}
    r[0] &= r[1] \ll 2; \\
    r[1] &= r[1] + 1;
\end{align*}
\]

L18

\[
\begin{align*}
    IC &= r[1] ? M[_n]; \\
    PC &= IC < 0 \rightarrow L19;
\end{align*}
\]
Example Sequence of Optimizations (cont.)

⇒ after code motion

```c
r[2] = 0;
r[1] = 0;
r[4] = M[_n];
PC = L18
L19
r[0] = r[1] << 2;
r[1] = r[1] + 1;
L18
IC = r[1] ? r[4];
PC = IC < 0 → L19;
```