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
  - use less energy
- An optimization phase consists of a sequence of code-improving transformations of the same type.

Types of Compiler Optimizations

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

Function Call Optimizations

- Procedure integration or inlining
  - Replaces a call with the body of the function being invoked.
- Procedure specialization or cloning
  - Makes specific copies of functions based on parameters.
- Tail call and recursion elimination
  - Eliminates calls at the end of a function.
- Function memoization
  - Uses a software cache to remember results of a function based on its input values.

Cloning

- Creates copies of a function, where each copy has different constant arguments. Enables other code improving transformations with less code growth than inlining.

```
    b = f(a, 2);
    ...
    int f(int x, int factor) {
       return x*factor;
    }
=>
    b = f_2(a);
    ...
    int f_2(int x) {
       return x<<1;
    }
```

Tail Recursion Elimination

- A function is tail recursive if it calls itself just before returning. The recursive call can be replaced with a jump to the top of the function.

```
int inarray(int a[], int x, int i, int n) {
   if (i == n) return false;
   else if (a[i] == x) return true;
   else return inarray(a, x, i+1, n);
}
=>
int inarray(int a[], int x, int i, int n) {
top:if (i == n) return false;
    else if (a[i] == x) return true;
    else {  i++;  goto top; }
}
```

Loop Optimizations

- Loop invariant code motion
  - Moves invariant computations out of a loop.
- Loop strength reduction
  - Used to step through array elements with additions.
- Induction variable elimination
  - Eliminates increments to loop variables.
- Loop unrolling
  - Reduces loop overhead by duplicating the loop body.
- Loop collapsing
  - Transforms a loop nest into a single loop to reduce loop overhead.

Loop Optimizations (cont.)

- Loop fusion
  - Merges multiple loops together to reduce loop overhead.
- Software pipelining
  - Reschedules a loop using code duplication so that different instructions from different original iterations are in the loop body.

Loop Unrolling

- Reduces loop overhead by duplicating the loop body when the number of iterations is known.

```
   for (i=0; i < n; i++)
      a[i] = b[i]+c[i];
=>
   if (0 < n) {
      for (i=0; i < n%4; i++)
         a[i] = b[i]+c[i];
      for (; i < n; i += 4) {
         a[i] = b[i]+c[i];
         a[i+1] = b[i+1]+c[i+1];
         a[i+2] = b[i+2]+c[i+2];
         a[i+3] = b[i+3]+c[i+3];
      }
   }
```

Loop Collapsing

- Combines a loop nest into a single loop. Can reduce loop overhead.

```
   int a[100][200];
   ...
   for (i = 0; i < 100; i++)
      for (j = 0; j < 200; j++)
         a[i][j] = 0;
=>
   for (i = 0; i < 20000; i++)
      a[i] = 0;
```

Loop Fusion

- Merges distinct loops to reduce loop overhead.

```
   for (i = 0; i < 100; i++)
      a[i] = 0;
   for (i = 0; i < 200; i++)
      b[i] = c[i];
=>
   for (i = 0; i < 100; i++) {
      a[i] = 0;
      b[i] = c[i];
   }
   for (i = 100; i < 200; i++)
      b[i] = c[i];
```

Memory Access Optimizations

- Register allocation
  - Replaces references to local variables and arguments with registers.
- Memory hierarchy improvement
  - Array padding
    - Adds extra elements within or at the end of arrays.
  - Scalar replacement
    - Replaces an array element with a scalar within a loop.
  - Loop interchange
    - Interchanges loop statements in a loop nest.
  - Prefetching
    - Special instructions are used to fetch data before it is needed to avoid cache misses or reduce cache delays.

Array Padding

- Unused data locations are inserted between arrays or within arrays. Can be used to reduce conflict misses in a cache or memory bank conflicts.

```
   double a[1024], b[1024];
   ...
   for (i = 0; i < 1024; i++)
      sum += a[i]*b[i];
=>
   double a[1024], pad[8], b[1024];
   ...
   for (i = 0; i < 1024; i++)
      sum += a[i]*b[i];
```

Scalar Replacement

- Replaces a loop-invariant array element with a scalar in a loop. Scalars can more easily be allocated to registers.

```
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
      total[i] = total[i]+a[i][j];
=>
for (i = 0; i < n; i++) {
   T = total[i];
   for (j=0; j < n; j++)
      T = T+a[i][j];
   total[i] = T;
}
```

Loop Interchange

- Changes the position of two loop statements in a perfect loop nest. Often used to improve spatial locality.

```
   for (j = 0; j < n; j++)
      for (i = 0; i < m; i++)
         total += a[i][j];
=>
   for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
         total += a[i][j];
```

Control Flow Optimizations

- Jump elimination
  - Branch chaining
    - Avoid jumping to a location that has an unconditional jump.
  - Reversing branches
    - Reverses the sense of a conditional branch over a jump.
  - Code positioning
    - Eliminates an unconditional jump by moving its target to follow the jump when the target has a single predecessor.
  - Loop inversion
    - Places a loop exit test at the bottom of a loop instead of the top.
  - Useless jump elimination
    - Eliminates jumps to a block that immediately follows the jump.

Control Flow Optimizations (cont.)

- Unreachable code elimination
  - Eliminates code that cannot possibly be executed.

Data Flow Optimizations

- Common subexpression elimination
  - Eliminates fully redundant computations.
- Partial redundancy elimination
  - Applies CSE along specific paths.
- Dead assignment elimination
  - Eliminates assignments to destinations that are never used.
- Evaluation order determination
  - Reorders operations to require fewer registers.
- Recurrence elimination
  - Avoids redundant loads across loop iterations.

Recurrence Elimination

- Avoids redundant memory loads across loop iterations. The scalar variable *v* will likely be later allocated to a register.

```
   for (i = 1; i < n; i++)
      a[i] = a[i] + a[i-1];
=>
   v = a[0];
   for (i = 1; i < n; i++) {
      v = a[i] + v;
      a[i] = v;
   }
```

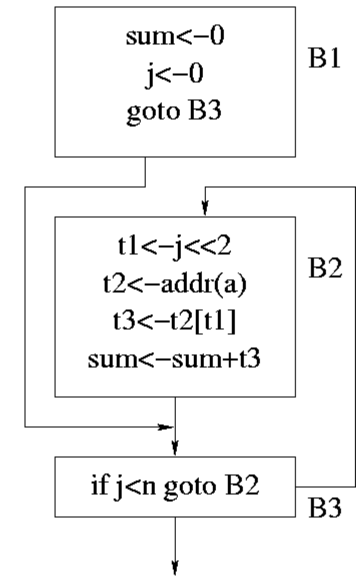
Machine-Specific Optimizations

- Instruction scheduling
  - Reorders instructions to avoid pipeline stalls.
- Filling delay slots
  - Places instructions after transfers of control when the effect of the transfer of control does not occur until after the instruction.
- Exploiting instruction-level parallelism
  - Exploits architectural features (e.g. VLIW) to schedule multiple operations to be issued in parallel.
- Peephole optimization (includes instruction selection)
  - Applies improvements to a small window of instructions.

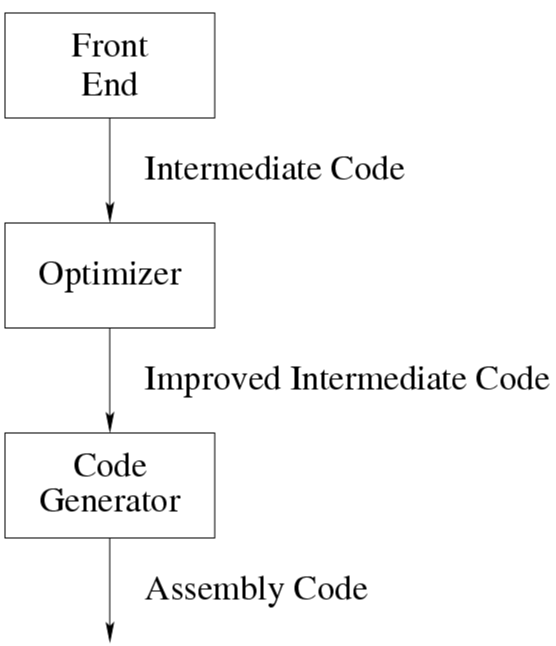
Control Flow

- Basic block - a sequence of consecutive statements with exactly 1 entry and 1 exit
- Control Flow graph - a directed graph where the nodes are basic blocks and block $B_1 \rightarrow$ block B_2 iff B_2 can be executed immediately after B_1
- Local optimizations - performed only within a basic block
- Global optimizations - performed across basic blocks

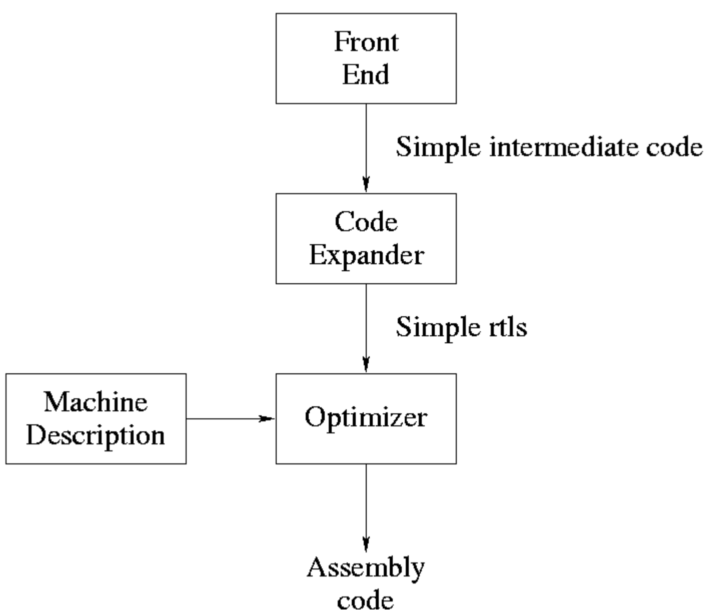
Example Control Flow Graph



Optimizations before Code Generation



Optimizations after Code Generation



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]];  r[1]:
     ⇒
          r[2]=M[r[30]+i]; r[1]=r[30]+i; r[1]:
     or
          r[2]=M[r[30]+i];   r[1]:
```

Combining Three RTLs

```
31        r[2]=M[r[3]];
32 {31}   r[2]=r[2]+1;
33 {32}   M[r[3]]=r[2];  r[2]:
     ⇒
          M[r[3]]=M[r[3]]+1; r[2]=M[r[3]]+1;  r[2]:
     or
          M[r[3]]=M[r[3]]+1;  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]];      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 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 r[36]=r[5];
39 {38} r[36]=r[36]+i;

42 **r[40]=M[r[5]+i]];**
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 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]; r[41]:
46 {45,39} M[r[36]]=r[42]; r[42]:r[36]:

Cascading Instruction Selection (cont.)

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

44 r[42]=M[r[5]+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          r[36]=r[5];

44          r[42]=M[r[5]+i]];
45 {44}     r[42]=r[42]+1;
46 {45,38}  M[r[36]+i]=r[42];   r[42]:r[36]:
```

Cascading Instruction Selection (cont.)

```
44          r[42]=M[r[5]+i]];
45 {44}     r[42]=r[42]+1;
46 {45}     M[r[5]+i]=r[42];    r[42]:
```

Cascading Instruction Selection (cont.)

```
            M[r[5]+i]=M[r[5]+i]+1;
```

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;
   M[r[13]+sum]=M[r[13]+sum]+M[r[0]+_a];
   M[r[13]+j] = M[r[13]+j]+1;
L18
   IC=M[r[13]+j]?M[_n];
   PC=IC<0→L19;
```

Example Sequence of Optimizations (cont.)

⇒ after register allocation

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

Example Sequence of Optimizations (cont.)

⇒ after loop-invariant code motion

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

Example Sequence of Optimizations (cont.)

⇒ after loop strength reduction

```
  r[2]=0;
  r[1]=0;
  r[4]=M[_n];
  r[3]=_a;
  PC=L18;
L19
  r[0]=r[1]<<2;
  r[2]=r[2]+M[r[3]];
  r[3]=r[3]+4;
  r[1]=r[1]+1;
L18
  IC=r[1]?r[4];
  PC=IC<0→L19;
```

Example Sequence of Optimizations (cont.)

⇒ after dead assignment elimination

```
  r[2]=0;
  r[1]=0;
  r[4]=M[_n];
  r[3]=_a;
  PC=L18;
L19
  r[2]=r[2]+M[r[3]];
  r[3]=r[3]+4;
  r[1]=r[1]+1;
L18
  IC=r[1]?r[4];
  PC=IC<0→L19;
```

Example Sequence of Optimizations (cont.)

⇒ after basic induction variable elimination

```
  r[2]=0;
  r[1]=0;
  r[4]=M[_n]<<2;
  r[3]=_a;
  r[4]=r[4]+r[3];
  PC=L18;
L19
  r[2]=r[2]+M[r[3]];
  r[3]=r[3]+4;
L18
  IC=r[3]?r[4];
  PC=IC<0→L19;
```

Example Sequence of Optimizations (cont.)

⇒ after dead assignment elimination

```
  r[2]=0;
  r[4]=M[_n]<<2;
  r[3]=_a;
  r[4]=r[4]+r[3];
  PC=L18;
L19
  r[2]=r[2]+M[r[3]];
  r[3]=r[3]+4;
L18
  IC=r[3]?r[4];
  PC=IC<0→L19;
```

Example of Common Subexpression Elimination

```
  r[1] = M[r[13] + i] << 2;
  r[1] = M[r[1] + _b];
  r[2] = M[r[13] + i] << 2;
  r[2] = M[r[2] + _b];
⇒
  r[1] = M[r[13] + i] << 2;
  r[1] = M[r[1] + _b];
  r[2] = r[1];
```

Example of Unreachable Code Elimination

```
  ...
  PC = L12;
  r[1] = M[r[13] + i];
  r[1] = r[5] + r[1];
  M[r[13] + j] = r[1];
L13
  ...
⇒
  ...
  PC = L12;
L13
  ...
```

Example of Branch Chaining

```
IF a THEN b
```

compiles into

```
     a
     PC=IC!=0→L1;
     b
L1:
```

```
WHILE c DO d
```

compiles into

```
L2:  c
     PC=IC!=0→L3;
     d
     PC=L2;
L3:
```

Example of Branch Chaining (cont.)

```
WHILE c DO
   IF a THEN b
```

compiles into

```
L2:  c
     PC=IC!=0→L3;
     a
     PC=IC!=0→L1;
     b
L1:  PC=L2;
L3:
```

Example of Branch Chaining (cont.)

⇒ after branch chaining

```
L2:  c
     PC=IC!=0→L3;
     a
     PC=IC!=0→L2;
     b
L1:  PC=L2;
L3:
```

Example of Jump Elimination by Reversing Branches

```
     PC=IC==0→L1;
     PC=L2;
L1:
```

⇒

```
     PC=IC!=0→L2;
L1:
```

Example of Instruction Scheduling

```
    r[2]=M[r[30]+j];
    M[r[30]+k]=r[2];
    r[1]=r[1]+1;
⇒
    r[2]=M[r[30]+j];
    r[1]=r[1]+1;
    M[r[30]+k]=r[2];
```

Filling Delay Slot Example

```
    r[2]=M[a];
    IC=r[3]?0;
    PC=IC<0→L5;
    NL=NL;
⇒
    IC=r[3]?0;
    PC=IC<0→L5;
    r[2]=M[a];
```