The Lecture Contains:

- Code Optimization
- Most Important Optimizations
- Low Level Model of Optimization
- Mixed Model of Optimization
- Placement of Optimizations
- Placement of Optimizations: Continued
- Common Subexpression Elimination
- Copy Propagation
- Dead Code Elimination
- Algebraic Transformation
- Loop Optimizations
- Loop-unrolling
- Induction Variable Simplification
- Loop Jamming

Previous Next

Code Optimization

int a,b,c,d	ldw a,r1	add r1,r2,r3
c = a+b	ldw b,r2	add r3,1,r4
d = c+1	add r1,r2,r3	
	stw r3,c	
	ldw c,r3	
	add r3,1,r4	
	stw r4,d	
source code	naive sparc code	optimized code
	10 cycles	2 cycles

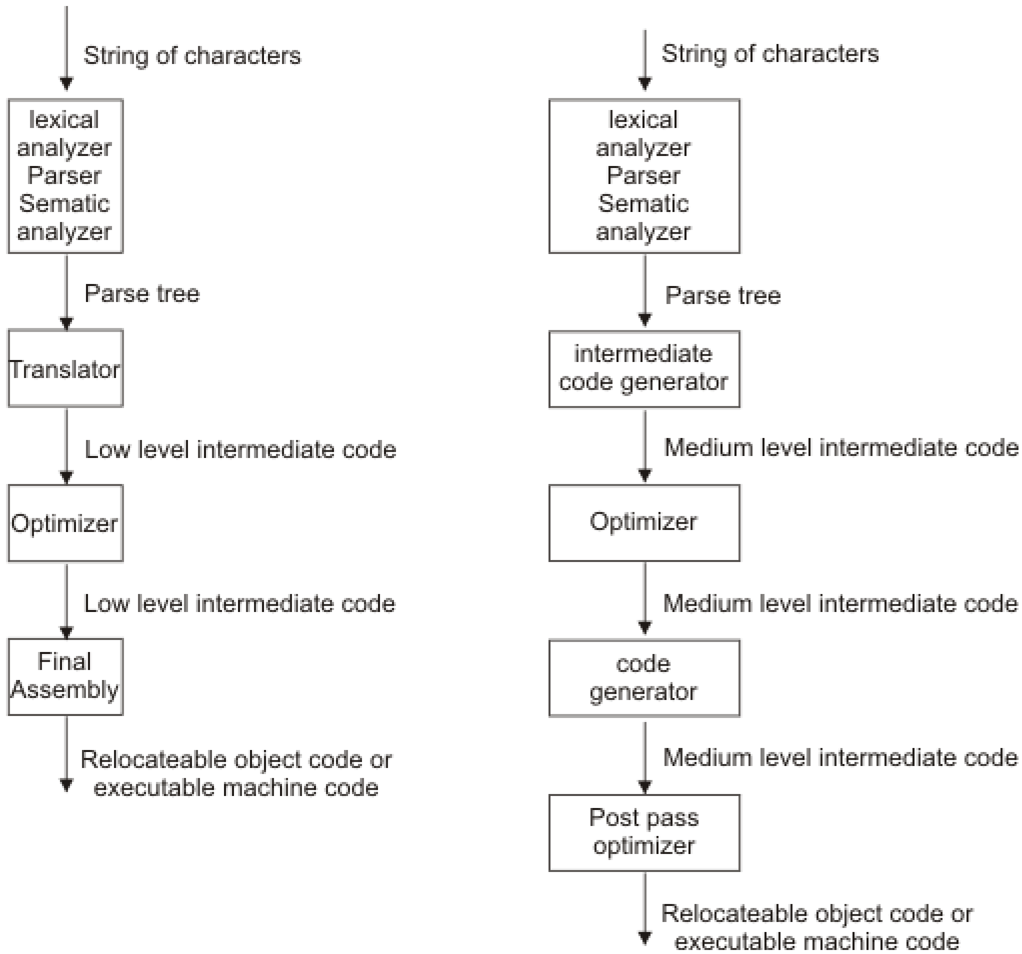
Most Important Optimizations

- Loop Optimizations
  - Moving loop invariant computations
  - Simplifying or eliminating computations on induction variables
- Global register allocation
- Instruction scheduling
- Some optimizations may be relevant to a particular program and may vary according to the structure and details of the programs
  - A highly recursive program may benefit from tail call optimization
  - A program with few loops and large basic blocks may benefit from loop distribution
  - In-lining may decrease call over heads; in-lining may increase the code size and may have negative effect on performance by increasing cache misses

Previous Next

Module 12: View

Lecture 24: Code Optimization



Choice is largely one of investment and development focus.

Low Level Model of Optimization

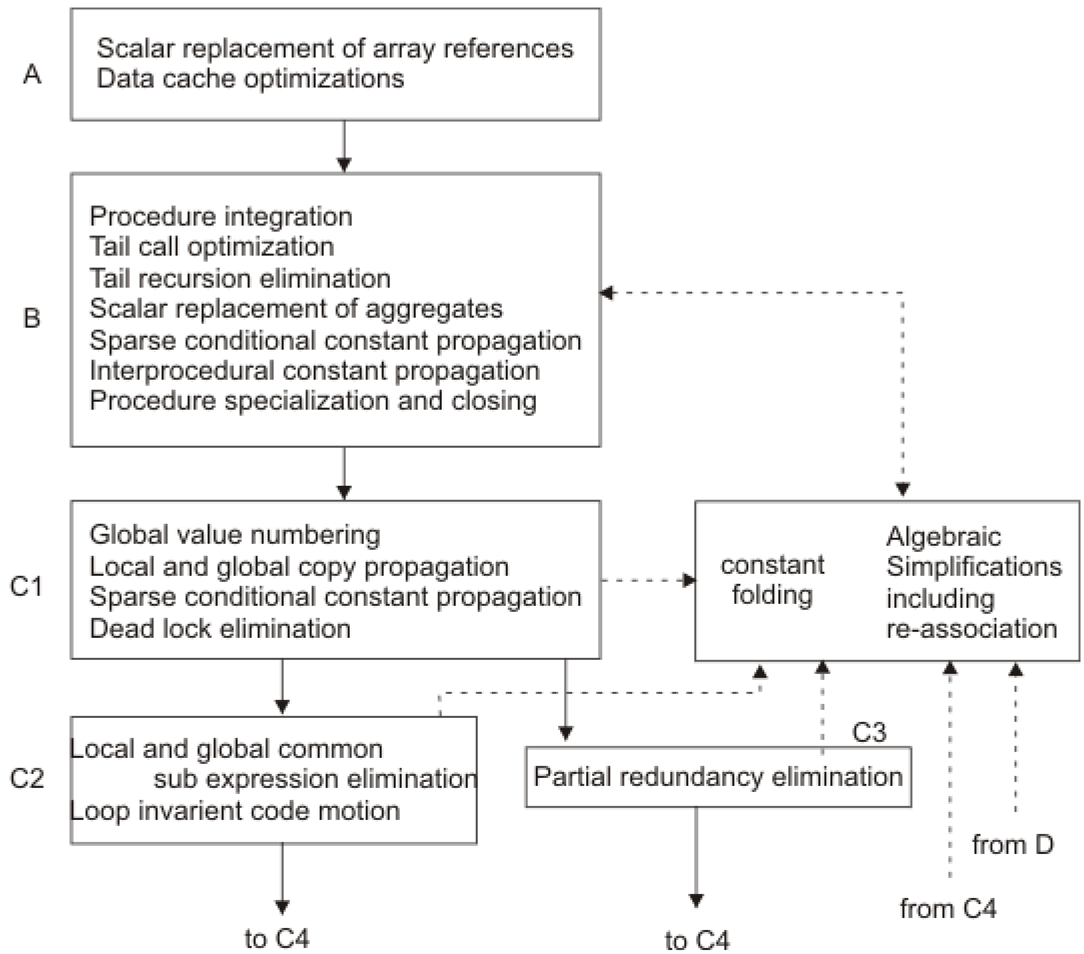
- Difficult to port unless second architecture is close to the first one
- Used in IBM Power-PC and HP PA-RISC
- Easier to avoid phase ordering problem
- Exposes all addresses to the optimizer
- Recommended for similar architectures

Previous Next

Mixed Model of Optimization

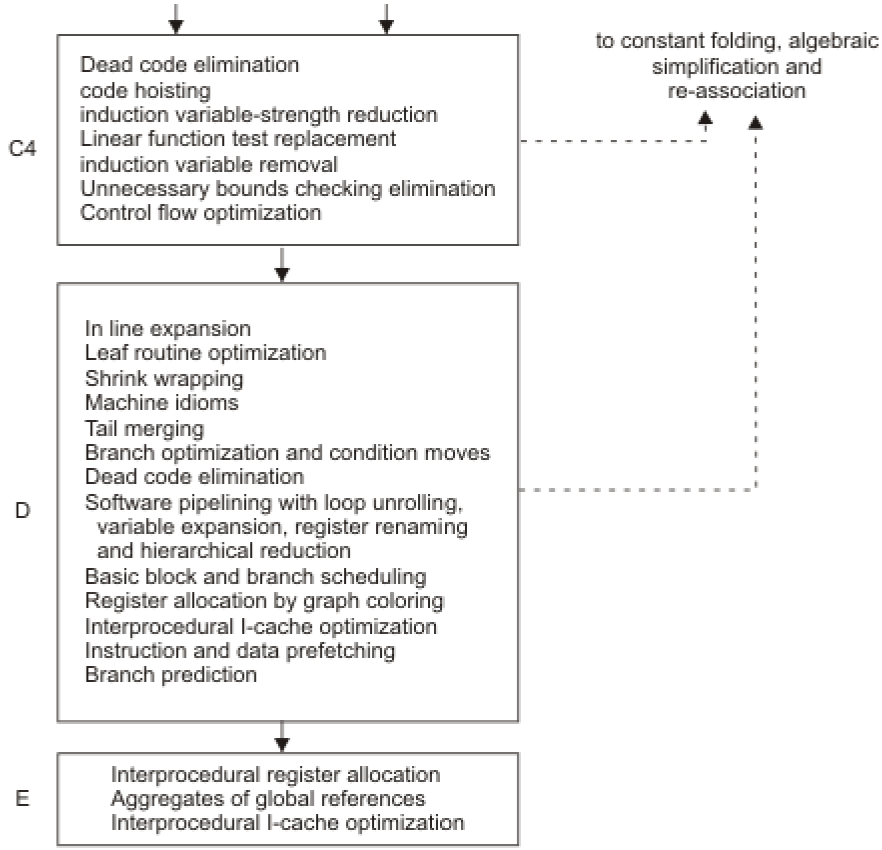
- More easily to be adapted to new architectures
- May be more efficient at compile time
- Used in Sun Sparc, Digital Alpha, Intel, SGI MIPS
- More appropriate for same language and many architectures

Placement of Optimizations



Previous Next

Placement of Optimizations: Cont'd



Code Optimization

Criteria for Code Improving Transformation

- Preserve the meaning
- Must speed up the program
- Must be worth the effort
- The analysis must be fast

Local transformation : Within basic blocks
Global transformation : Across basic blocks

Previous Next

Module 12: View

Lecture 24: Code Optimization

Common Subexpression Elimination

X := a + b Y := a + b	⇒	X := a + b Y := X

t_6 := 4 * i X := a[t_6] t_7 := 4 * i t_8 := 4 * j t_9 := a[t_8] a[t_7] := t_9 t_{10} := 4 * j a[t_{10}] := X goto L	CSE ⇒ (local)	t_6 := 4 * i X := a[t_6] t_8 := 4 * j t_9 := a[t_8] a[t_6] := t_9 a[t_8] := X gotoL

t_6 := 4 * i X := a[t_6] t_8 := 4 * j t_9 := a[t_8] a[t_6] := t_9 a[t_8] := X gotoL	t_4:=4 * j already ⇒ computed	t_6 := 4 * i X := a[t_6] t_9 := a[t_4] a[t_6] := t_9 a[t_4] := X goto L

t_6 := 4 * i X := a[t_6] t_9 := a[t_4] a[t_6] := t_9 a[t_4] := X gotoL	t_9 computes ⇒ a[j]	t_6 := 4 * i X := a[t_6] a[t_6] := t_5 a[t_4] := X gotoL

Previous Next

```
t6 := 4 * i
X := a[t6]                valueXcontains a[i]      X := t3
a[t6] := t5                      =>                a[t2] := t5
a[t4] := X                     use t3              a[t4] := X
goto L                                             goto L
```

Copy Propagation

Use g for f after assignment f = g

```
X := t3              X := t3
a[t2] := t5          a[t2] = t5
a[t4] := X     =>    a[t4] := t3
goto L               gotoL
```

```
X := Y                     X := Y
if X > n gotoL      =>     if Y > n gotoL
```

Dead Code Elimination

Dead Operation : Unreachable by any path produces a value not used

- If whose true and false arcs are same
- If whose B expr known at compile time
  loop not to be executed
  procedure not to be called

debug := false
...
if (debug) {
...
}

```
X := t3
a[t2] := t5     =>     a[t2] := t5
a[t4] := t3            a[t4] := t3
goto L                 gotoL
```

Previous Next

Renaming Temporary Variable Rename temporary variable t to u and replace all the occurrences of
t by u.
This transformation increases parallelism.
Interchange Statements Two statements may be interchanged if value of the block is not affected.

t1 = b + c	t2 = X + Y
t2 = X + Y	t1 = b + c

Constant folding Evaluate constant expressions at compile time.

X = 3 + 5	X = 8
Y = X * 2	Y = 16

Algebraic Transformation

- Eliminate addition/subtraction with 0 X = X ± 0 should be eliminated.
- Eliminate multiplication/division by 1 X = X * 1 or X = X/1 should be eliminated.
- Eliminate multiplication by 0 X = X * 0 should be replaced with X = 0

Strength reduction Costly operators should be replaced by cheaper operators

- Replace T = X * * *2 by T = X * X
- Replace T = X * 4 by T = ls(X, 2)
- Replace 2 * X by X + X
- Replace X * 0.5 by X/2
- Replace X/2 by rs(X, 1)

Loop Optimizations

Code Motion: Expression not evaluated in the code must be moved out of loop

```
while( i <= limit - 2 ) {
\* statement not changing limit *\
}
```

$$\Downarrow$$

```
t := limit - 2
while( i <= t ) { statement }
```

$X := t_3$ $a[t_2] := t_5$ $a[t_4] := t_3$ goto L	$\Rightarrow$	$a[t_2] := t_5$ $a[t_4] := t_3$ gotoL

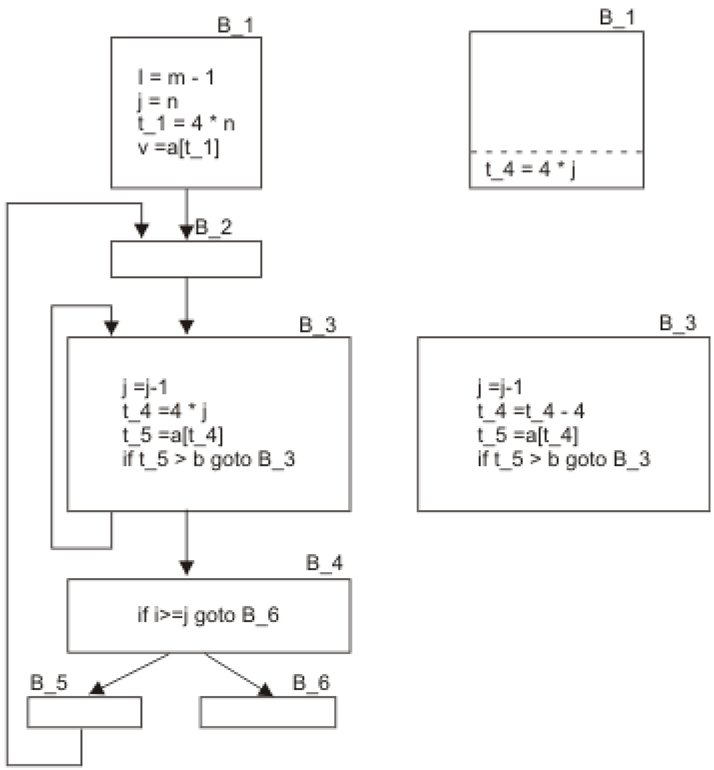
Previous Next

Loop-unrolling

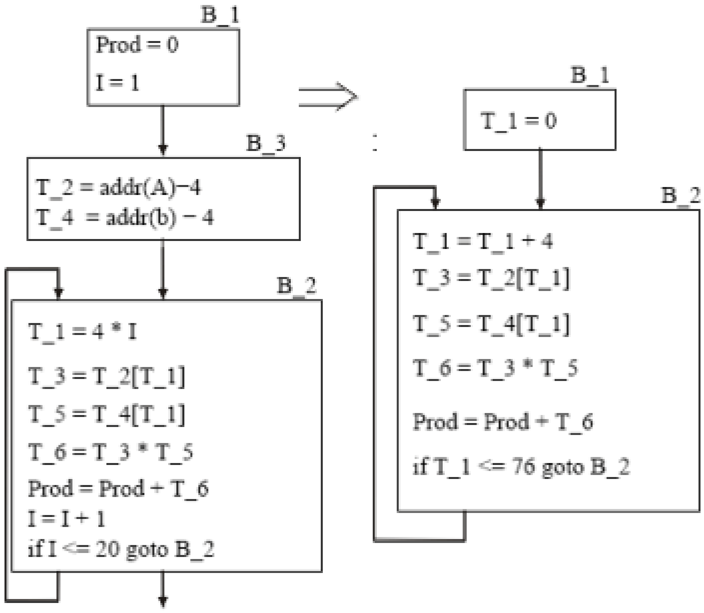
i := 1
loop
  if i > n then exitloop
  body
exit
	$\Rightarrow$	i:=1
		loop
		  if i > n then exitloop
		  $body_0$
		if i > n then exitloop
		  $body_1$
		    $\vdots$
		if i > n then exitloop
		  $body_{n-1}$
		exit

DO I = 1 to 100 by 1 A(I) = A(I) + B(I) END	$\Longrightarrow$	DO I = 1 to 100 by 2 A(I) = A(I) + B(I) A(I+1) = A(I+1) + B(I+1) END

Induction Variable Simplification



Induction Variable Simplification



Loop Jamming

Two adjacent loops may me merged into a single loop

```
For I = 1 to 100
      A(I) = 0
Endfor
For I = 1 to 100
      B(I) = X(I) + Y
Endfor
```

can be replaced by

```
For I = 1 to 100
      A(I) = 0
      B(I) = X(I) + Y
Endfor
```

Previous   Next