Local and Global Optimizations

Y.N. Srikant

(Formerly) Professor
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

ACM Winter School on Design, Implementation
and Verification of Computer Systems
January 3-16, 2022.

Outline of the Lecture

- What is code optimization and why is it needed?
- Types of optimizations
- Basic blocks and control flow graphs
- Local optimizations
- Directed acyclic graphs and value numbering
- Examples of global optimizations

Machine-independent Code Optimization

- Intermediate code generation process introduces many inefficiencies.
    - Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.
- Code optimization removes such inefficiencies and improves code.
- Improvement may be time, space, or power consumption.

Machine-independent Code Optimization

- It changes the structure of programs, sometimes of beyond recognition.
    - Inlines functions, unrolls loops, eliminates some programmer-defined variables, etc.
- Code optimization consists of a bunch of heuristics and percentage of improvement depends on programs (may be zero also).
- Optimizations may be classified as *local* and *global*.

Local Optimizations

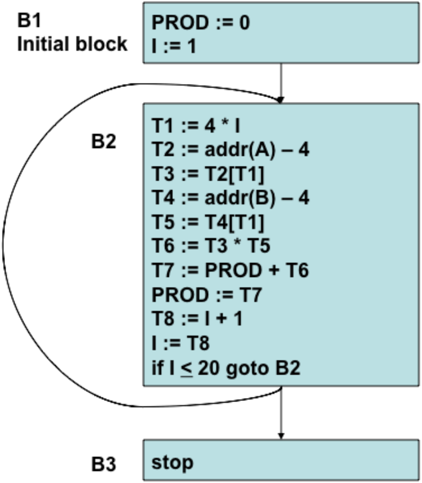
Local optimizations: within basic blocks

- Local common subexpression elimination.
- Dead code (instructions that compute a value that is never used) elimination.
- Reordering computations using algebraic laws.
- Peephole optimizations.

Basic Blocks and Control-Flow Graphs

- Basic blocks are sequences of intermediate code with a *single* entry and a *single* exit.
- We consider the quadruple version of intermediate code here, to make the explanations easier.
- Control flow graphs show flow of control among basic blocks.
- Basic blocks are represented as *directed acyclic blocks*(DAGs), which are in turn represented using the value-numbering method applied on quadruples.

Example of Basic Blocks and Control Flow Graph



B1
Initial block

```
PROD := 0
I := 1
```

B2

```
T1 := 4 * I
T2 := addr(A) – 4
T3 := T2[T1]
T4 := addr(B) – 4
T5 := T4[T1]
T6 := T3 * T5
T7 := PROD + T6
PROD := T7
T8 := I + 1
I := T8
if I ≤ 20 goto B2
```

B3

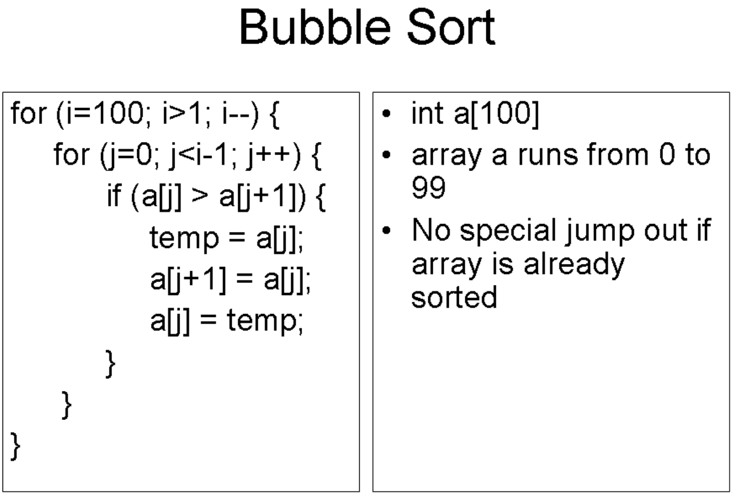
```
stop
```

High level language code:

```
{ PROD = 0;
  for ( I = 1; I <= 20; I++)
    PROD = PROD + A[I] * B[I];
}
```

```
PROD := 0
I := 1
T1 := 4 * I
T2 := addr(A) – 4
T3 := T2[T1]
T4 := addr(B) – 4
T5 := T4[T1]
T6 := T3 * T5
T7 := PROD + T6
PROD := T7
T8 := I + 1
I := T8
if I ≤ 20 goto B2
stop
```

Bubble Sort

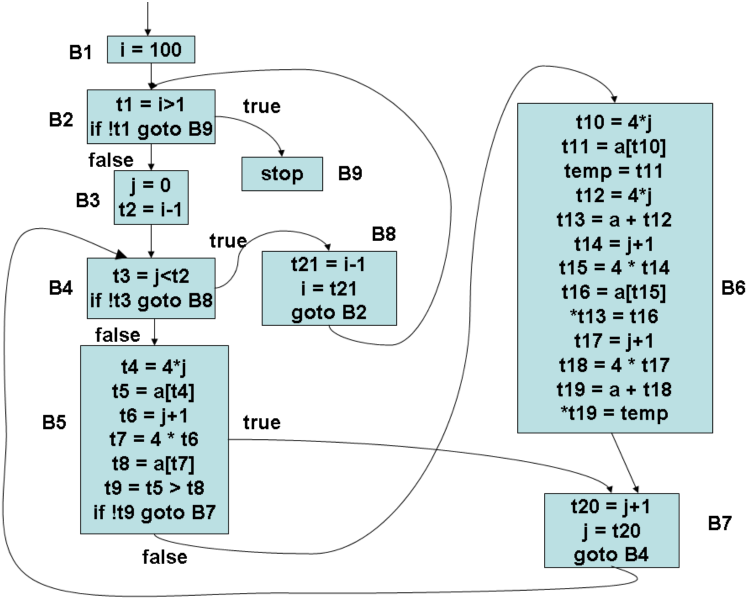


Bubble Sort

```
for (i=100; i>1; i--) {
    for (j=0; j<i-1; j++) {
        if (a[j] > a[j+1]) {
            temp = a[j];
            a[j+1] = a[j];
            a[j] = temp;
        }
    }
}
```

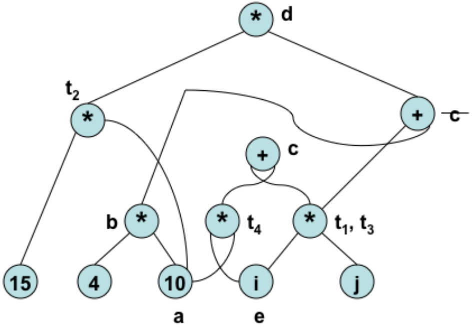
- int a[100]
- array a runs from 0 to 99
- No special jump out if array is already sorted

Control Flow Graph of Bubble Sort



Example of a Directed Acyclic Graph (DAG)

1. $a = 10$
2. $b = 4 * a$
3. $t1 = i * j$
4. $c = t1 + b$
5. $t2 = 15 * a$
6. $d = t2 * c$
7. $e = i$
8. $t3 = e * j$
9. $t4 = i * a$
10. $c = t3 + t4$



Value Numbering in Basic Blocks

- A simple way to represent DAGs is via *value-numbering*.
- While searching DAGs represented using pointers etc., is inefficient, *value-numbering* uses hash tables and hence is very efficient.
- Central idea is to assign numbers (called value numbers) to expressions in such a way that two expressions receive the same number if the compiler can prove that they are equal for all possible program inputs.
- We assume quadruples with binary or unary operators.
- The algorithm uses three tables indexed by appropriate hash values:
  HashTable, ValnumTable, and *NameTable*.
- Can be used to eliminate common sub-expressions, do constant folding, and constant propagation in basic blocks.
- Can take advantage of commutativity of operators, addition of zero, and multiplication by one.

Data Structures for Value Numbering

In the field *Namelist*, first name is the defining occurrence and replaces all other names with the same value number with itself (or its constant value)

HashTable entry
(indexed by expression hash value)

Expression	Value number

ValnumTable entry
(indexed by name hash value)

Name	Value number

NameTable entry
(indexed by value number)

Name list	Constant value	Constflag

Example of Value Numbering

HLL Program	Quadruples before Value-Numbering	Quadruples after Value-Numbering
$a = 10$	1. $a = 10$	1. $a = 10$
$b = 4 * a$	2. $b = 4 * a$	2. $b = 40$
$c = i * j + b$	3. $t1 = i * j$	3. $t1 = i * j$
$d = 15 * a * c$	4. $c = t1 + b$	4. $c = t1 + 40$
$e = i$	5. $t2 = 15 * a$	5. $t2 = 150$
$c = e * j + i * a$	6. $d = t2 * c$	6. $d = 150 * c$
	7. $e = i$	7. $e = i$
	8. $t3 = e * j$	8. $t3 = i * j$
	9. $t4 = i * a$	9. $t4 = i * 10$
	10. $c = t3 + t4$	10. $c = t1 + t4$
		(Instructions 5 and 8 can be deleted)

Example: *HashTable* and *ValNumTable*

HashTable

Expression	Value-Number
$i * j$	5
$t1 + 40$	6
$150 * c$	8
$i * 10$	9
$t1 + t4$	11

ValNumTable

Name	Value-Number
a	1
b	2
i	3
j	4
$t1$	5
c	6,11
$t2$	7
d	8
e	3
$t3$	5
$t4$	10

Example: *NameTable*

NameTable

Name	Constant Value	Constant Flag
a	10	T
b	40	T
i,e		
j		
$t1,t3$		
$t2$	150	T
d		
c		

Handling Commutativity etc.

- When a search for an expression $i + j$ in *HashTable* fails, try for $j + i$.
- If there is a quad $x = i + 0$, replace it with $x = i$.
- Any quad of the type, $y = j * 1$ can be replaced with $y = j$.
- After the above two types of replacements, value numbers of x and y become the same as those of i and j, respectively.
- Quads whose LHS variables are used later can be marked as *useful*.
- All unmarked quads can be deleted at the end.

Peephole Optimizations

- Simple but effective local optimizations.
- Usually carried out on machine code, but intermediate code can also benefit from it.
- Examines a sliding window of code (*peephole*), and replaces it by a shorter or faster sequence, if possible.
- Each improvement provides opportunities for additional improvements.
- Therefore, repeated passes over code are needed.

Peephole Optimizations

- Elimination of redundant instructions.
- Removing unreachable code.
- Short-circuiting jumps over jumps.
- Algebraic simplifications.
- Strength reduction.
- Use of machine idioms.

Elimination of Redundant Loads and Stores

Basic block B

Load X, R0
{no modifications
to R0 or X here}
Store R0, X

Store instruction
can be deleted

Basic block B

Load X, R0
{no modifications
to X or R0 here}
Load X, R0

Second Load instr
can be deleted

Basic block B

Store R0, X
{no modifications
to X or R0 here}
Load X, R0

Load instruction
can be deleted

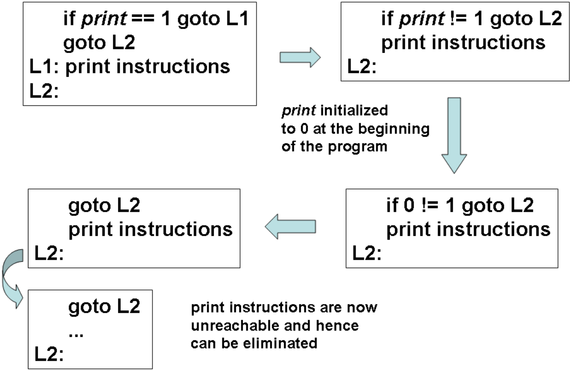
Basic block B

Store R0, X
{no modifications
to X or R0 here}
Store R0, X

Second Store instr
can be deleted

Removing Unreachable Code

- An unlabeled instruction immediately following an unconditional jump may be removed.
    - May be produced due to debugging code introduced during development.
    - Or due to updates to programs (changes for fixing bugs) without considering the whole program segment.



```
    if print == 1 goto L1
    goto L2
L1: print instructions
L2:
```

```
    if print != 1 goto L2
    print instructions
L2:
```

print initialized to 0 at the beginning of the program

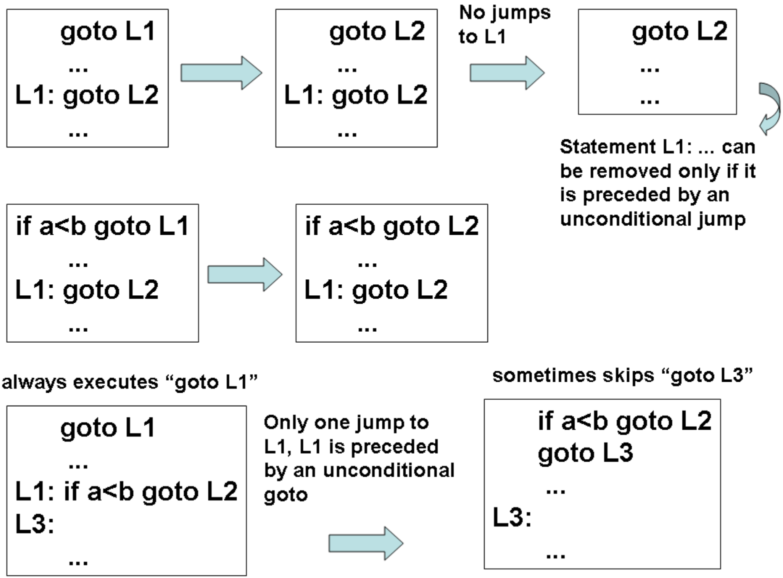
```
    if 0 != 1 goto L2
    print instructions
L2:
```

```
    goto L2
    print instructions
L2:
```

```
    goto L2
    ...
L2:
```

print instructions are now unreachable and hence can be eliminated

Short-circuiting Jumps over Jumps



```
      goto L1
        ...
L1: goto L2
        ...
```

```
      goto L2
        ...
L1: goto L2
        ...
```

No jumps to L1

```
      goto L2
        ...
        ...
```

Statement L1: ... can be removed only if it is preceded by an unconditional jump

```
if a<b goto L1
        ...
L1: goto L2
        ...
```

```
if a<b goto L2
        ...
L1: goto L2
        ...
```

always executes "goto L1"

```
      goto L1
        ...
L1: if a<b goto L2
L3:
        ...
```

Only one jump to L1, L1 is preceded by an unconditional goto

sometimes skips "goto L3"

```
      if a<b goto L2
      goto L3
        ...
L3:
        ...
```

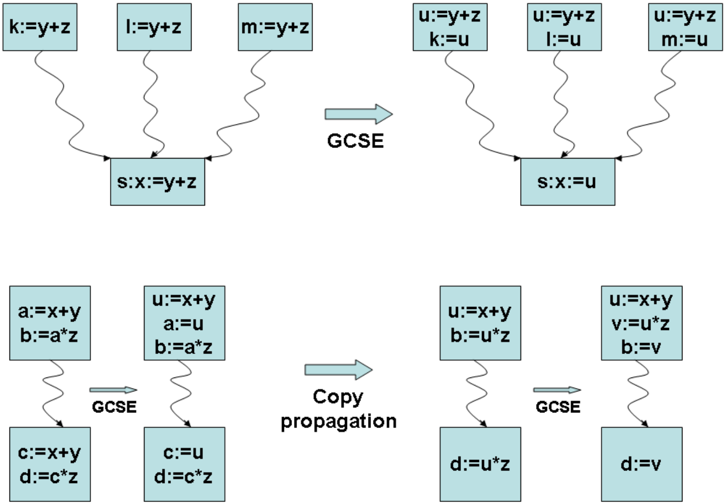
Reduction in Strength and Use of Machine Idioms

- x^2 is cheaper to implement as $x * x$ than as a call to an exponentiation routine.
- For integers, $x * 2^3$ is cheaper to implement as $x << 3$ (x left-shifted by 3 bits).
- For integers, $x/2^2$ is cheaper to implement as $x >> 2$ (x right-shifted by 2 bits).
- Floating point division by a constant c can be approximated as multiplication by its reciprocal, $1/c$. $1/c$ can be computed by the compiler.
- Auto-increment and auto-decrement addressing modes can be used wherever possible.
    - Subsume INCREMENT and DECREMENT operations (respectively).
- Detection of the Multiply-and-Add pattern is more complicated.

Examples of Global Optimizations

- Global common sub-expression elimination
- Copy propagation
- Constant propagation and constant folding
- Loop invariant code motion
- Induction variable elimination and strength reduction
- Partial redundancy elimination
- Dead code elimination
- Loop unrolling
- Function inlining
- Tail recursion removal
- Trace scheduling

GCSE Conceptual Example



k:=y+z

l:=y+z

m:=y+z

s:x:=y+z

GCSE

u:=y+z
k:=u

u:=y+z
l:=u

u:=y+z
m:=u

s:x:=u

a:=x+y
b:=a*z

c:=x+y
d:=c*z

GCSE

u:=x+y
a:=u
b:=a*z

c:=u
d:=c*z

Copy propagation

u:=x+y
b:=u*z

d:=u*z

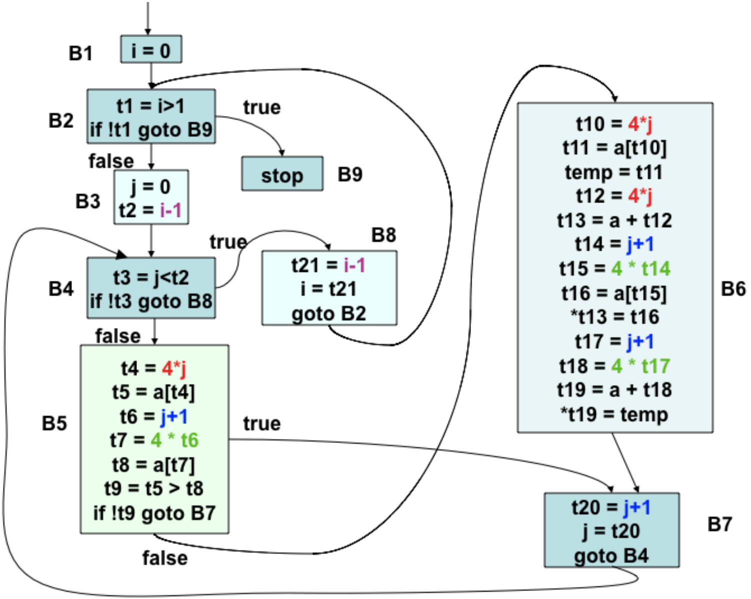
GCSE

u:=x+y
v:=u*z
b:=v

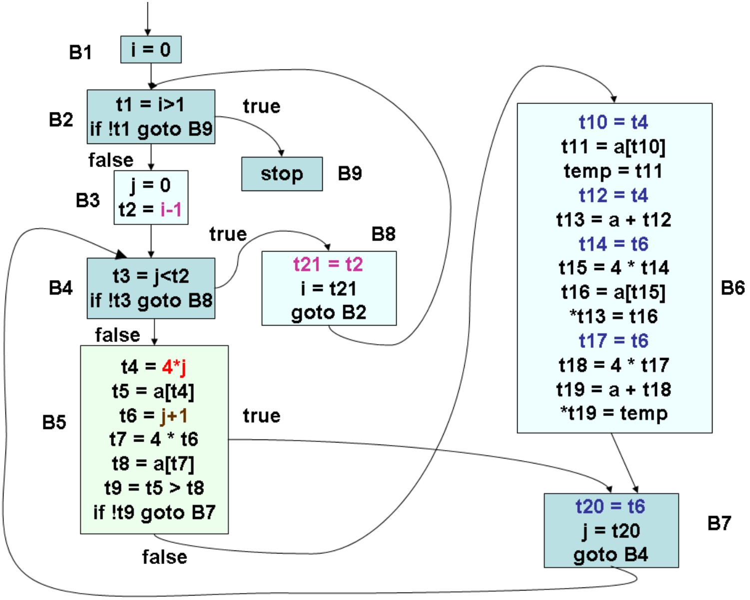
d:=v

Demonstrating the need for repeated application of GCSE

GCSE on Running Example - 1



GCSE on Running Example - 2



B1
i = 0

B2
t1 = i>1
if !t1 goto B9

B9
stop

B3
j = 0
t2 = i-1

B4
t3 = j<t2
if !t3 goto B8

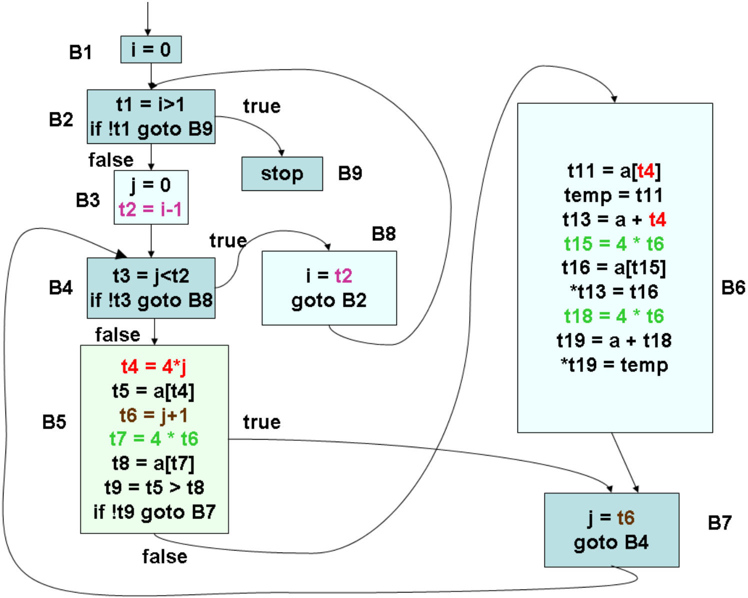
B8
t21 = t2
i = t21
goto B2

B5
t4 = 4*j
t5 = a[t4]
t6 = j+1
t7 = 4 * t6
t8 = a[t7]
t9 = t5 > t8
if !t9 goto B7

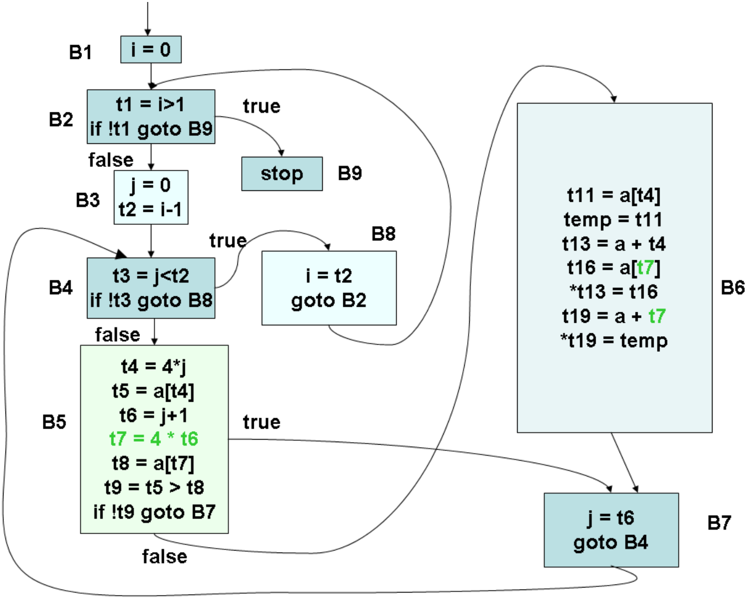
B6
t10 = t4
t11 = a[t10]
temp = t11
t12 = t4
t13 = a + t12
t14 = t6
t15 = 4 * t14
t16 = a[t15]
*t13 = t16
t17 = t6
t18 = 4 * t17
t19 = a + t18
*t19 = temp

B7
t20 = t6
j = t20
goto B4

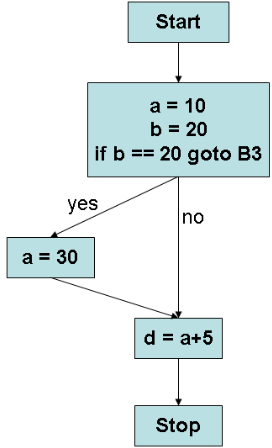
Copy Propagation on Running Example



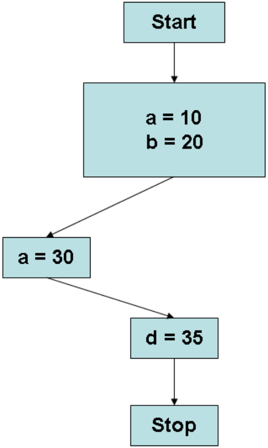
GCSE and Copy Propagation on Running Example



Constant Propagation and Folding Example

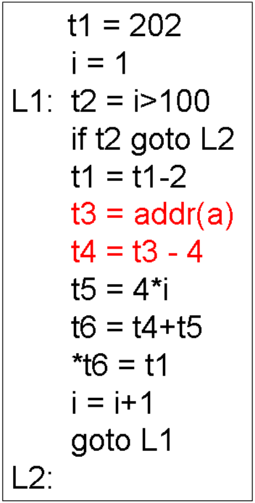


Before constant propagation



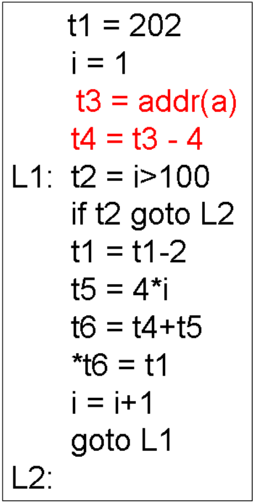
After constant propagation and folding

Loop Invariant Code motion Example



```
      t1 = 202
      i = 1
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t3 = addr(a)
      t4 = t3 - 4
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

Before LIV code motion



```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 - 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

After LIV code motion

Strength Reduction

```
       t1 = 202
       i = 1
       t3 = addr(a)
       t4 = t3 - 4
L1:    t2 = i>100
       if t2 goto L2
       t1 = t1-2
       t5 = 4*i
       t6 = t4+t5
       *t6 = t1
       i = i+1
       goto L1
L2:
```

Before strength reduction for t5

```
       t1 = 202
       i = 1
       t3 = addr(a)
       t4 = t3 – 4
       t7 = 4
L1:    t2 = i>100
       if t2 goto L2
       t1 = t1-2
       t6 = t4+t7
       *t6 = t1
       i = i+1
       t7 = t7 + 4
       goto L1
L2:
```

After strength reduction for t5 and copy propagation

Induction Variable Elimination

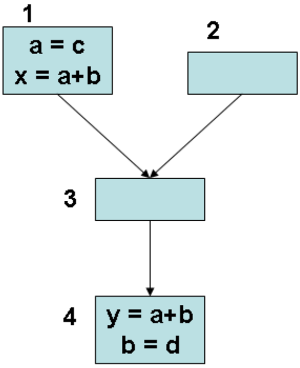
```
      t1  = 202
      i = 1
      t3 = addr(a)
      t4 = t3 – 4
      t7 = 4
L1:  t2 = i>100
      if t2 goto L2
      t1  = t1-2
      t6 = t4+t7
      *t6 = t1
      i = i+1
      t7 = t7 + 4
      goto L1
L2:
```

Before induction variable elimination (i)

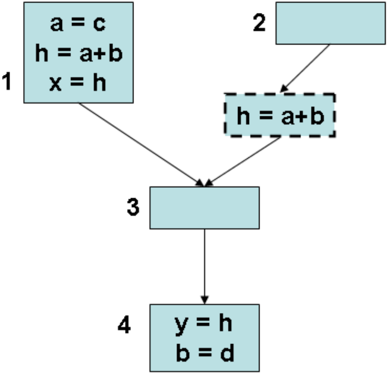
```
      t1  = 202
      t3 = addr(a)
      t4 = t3 – 4
      t7 = 4
L1:  t2 = t7 >400
      if t2 goto L2
      t1  = t1-2
      t6 = t4+t7
      *t6 = t1
      t7 = t7 + 4
      goto L1
L2:
```

After eliminating i and replacing it with t7

Partial Redundancy Elimination

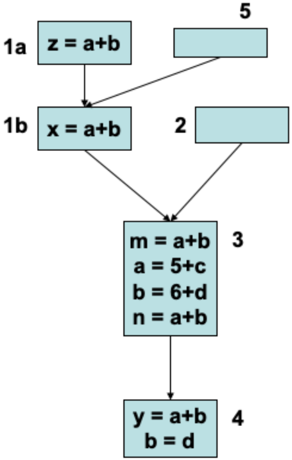


(a)

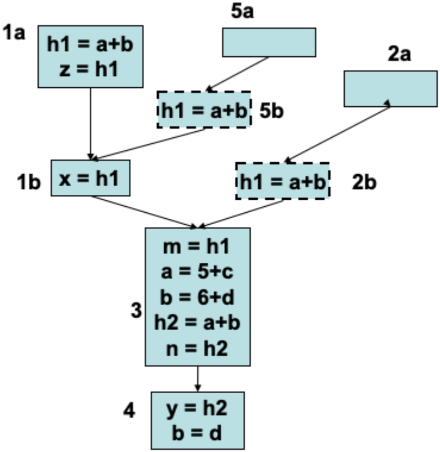


(b)

PRE Example 2

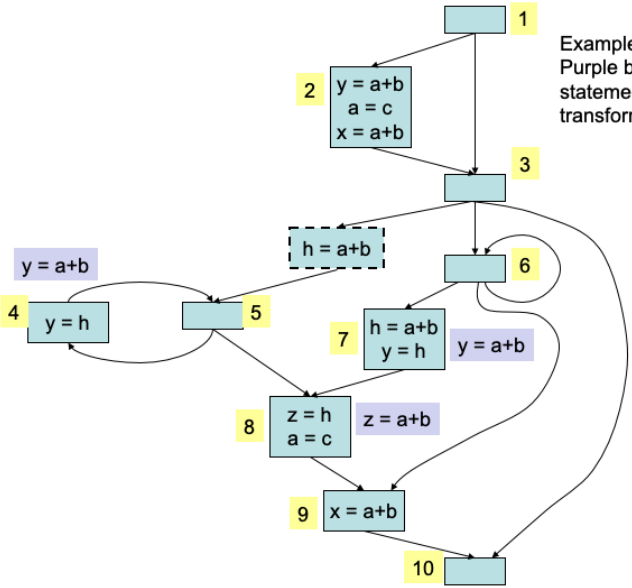


(a)



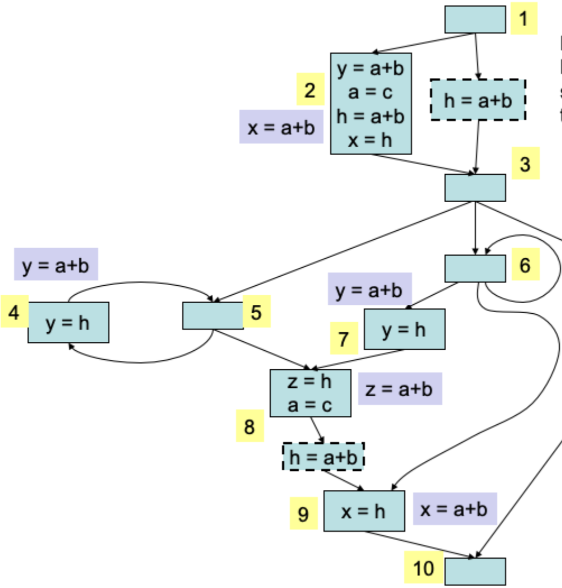
(b)

PRE Example 3



Example 3
Purple boxes show
statements before
transformation.

PRE Example 4



Example 4
Purple boxes show statements before transformation.

Dead Code Elimination - Easy Example

Code that is unreachable or that does not affect the program can be eliminated.

```
int g;
void f () { int i;
  i = 10; g = 100;     /* dead code */
  g = 250;
  return;
  g = 300;      /* unreachable code */
}
```

Code after optimization:

```
int g;
void f () {
  g = 250;
  return;
}
```

Dead Code Elimination - More Difficult Example

```
int foo(int x, int y) {
   int a = x + y;  /* useless code */
   if (x > 0)    /* useless code */
      a = 1;  /* useless code */
   return y;
}
```

Code after optimization:

```
int foo(int x, int y) {
   return y;
}
```

Unrolling a For-loop

```
for (i = 0; i<N; i++) { $S_1$(i); $S_2$(i); }

for (i = 0; i+3 < N; i+=3) {
    $S_1$(i); $S_2$(i);
    $S_1$(i+1); $S_2$(i+1);
    $S_1$(i+2); $S_2$(i+2);
}
// remaining few iterations, 1,2, or 3:
// (((N-1) mod 3)+1)
for (k=i; k<N; k++) { $S_1$(k); $S_2$(k); }
```

Unrolling While and Repeat loops

```
while (C) { $S_1$; $S_2$; }

while (C) {
    $S_1$; $S_2$;
    if (!C) break;
    $S_1$; $S_2$;
    if (!C) break;
    $S_1$; $S_2$;
}
```

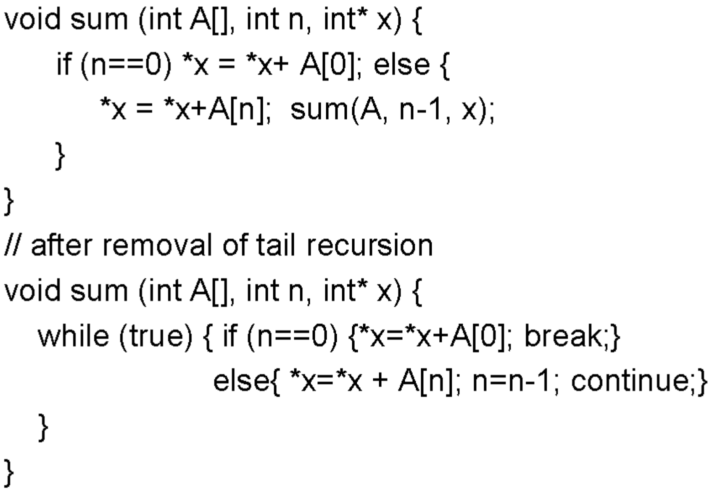
```
repeat { $S_1$; $S_2$; } until C;

repeat {
    $S_1$; $S_2$;
    if (C) break;
    $S_1$; $S_2$;
    if (C) break;
    $S_1$; $S_2$;
} until C;
```

Function Inlining

```
int find_greater(int A[10], int n) {  int i;
    for (i=0; i<10; i++){ if (A[i] > n) return i; }
}
// inlined call: x = find_greater(Y, 250);
int new_i, new_A[10];
new_A = Y;
for (new_i=0; new_i<10; new_i++) {
    if (new_A[new_i] > 250)
      { x = new_i; goto exit;}
}
exit:
```

Tail Recursion Removal



```
void sum (int A[], int n, int* x) {
     if (n==0) *x = *x+ A[0]; else {
          *x = *x+A[n];  sum(A, n-1, x);
     }
}
// after removal of tail recursion
void sum (int A[], int n, int* x) {
   while (true) { if (n==0) {*x=*x+A[0]; break;}
                  else{ *x=*x + A[n]; n=n-1; continue;}
   }
}
```

Trace Scheduling

- A Trace is a frequently executed acyclic sequence of basic blocks in a CFG (part of a path).
- Identifying a trace
  - Identify the most frequently executed basic block.
  - Extend the trace starting from this block, forward and backward, along most frequently executed edges.
- Apply list scheduling on the trace (including the branch instructions).
- Execution time for the trace may reduce, but execution time for the other paths may increase.
- However, overall performance will improve.

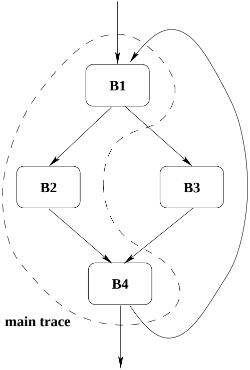
Trace Example

```
for (i=0; i < 100; i++)
{
   if (A[i] == 0)
     B[i] = B[i] + s;
   else
     B[i] = A[i];
   sum = sum + B[i];
}
```

(a) High-Level Code

		%% r1 ← 0
		%% r5 ← 0
		%% r6 ← 400
		%% r7 ← s
B1:	i1:	r2 ← load a(r1)
	i2:	if (r2 != 0) goto i7
B2:	i3:	r3 ← load b(r1)
	i4:	r4 ← r3 + r7
	i5:	b(r1) ← r4
	i6:	goto i9
B3:	i7:	r4 ← r2
	i8:	b(r1) ← r2
B4:	i9:	r5 ← r5 + r4
	i10:	r1 ← r1 + 4
	i11:	if (r1 < r6) goto i1

(b) Assembly Code



(c) Control Flow Graph

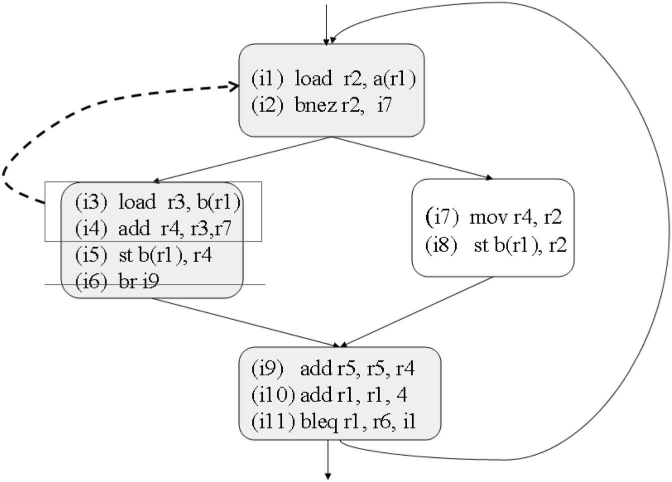
Trace - Basic Block Schedule

- 2-way issue architecture with 2 integer units.
- *add, sub, store*: 1 cycle, *load*: 2 cycles, *goto*: no stall.
- 9 cycles for the main trace and 6 cycles for the off-trace.

Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)		
1				
2	i2:	if (r2 != 0) goto i7		
3	i3:	r3 ← load b(r1)		
4				
5	i4:	r4 ← r3 + r7		
6	i5:	b(r1) ← r4	i6:	goto i9
3	i7:	r4 ← r2	i8:	b(r1) ← r2
7 (4)	i9:	r5 ← r5 + r4	i10:	r1 ← r1 + 4
8 (5)	i11:	if (r1 < r6) goto i1		

Trace Schedule

Trace Scheduling : Example



Trace Schedule

- 6 cycles for the main trace and 7 cycles for the off-trace.
- Speculative code motion - *load* instruction moved ahead of conditional branch
  - Example: Register r3 should not be live in block B3 (off-trace path).
  - May cause unwanted exceptions. Requires additional hardware support!

Time	Int. Unit 1		Int. Unit 2	
0	`i1:`	`r2 ← load a(r1)`	`i3:`	`r3 ← load b(r1)`
1				
2	`i2:`	`if (r2 != 0) goto i7`	`i4:`	`r4 ← r3 + r7`
3	`i5:`	`b(r1) ← r4`		
4 (5)	`i9:`	`r5 ← r5 + r4`	`i10:`	`r1 ← r1 + 4`
5 (6)	`i11:`	`if (r1 < r6) goto i1`		

3	`i7:`	`r4 ← r2`	`i8:`	`b(r1) ← r2`
4	`i12:`	`goto i9`		

Questions?