Code Optimization

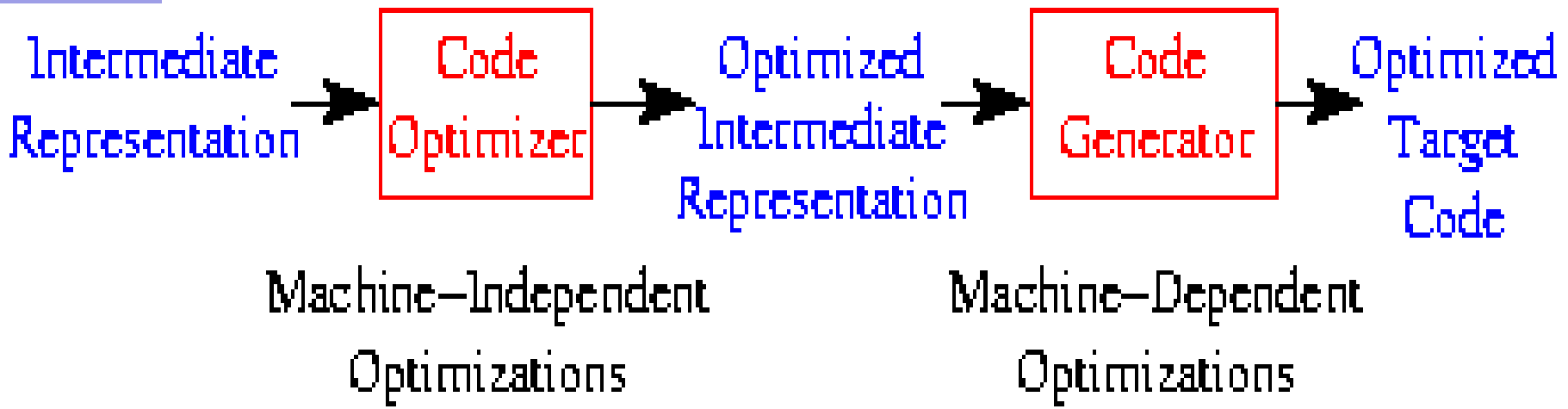
Code produced by compilation algorithms can often be improved (ideally optimized) in terms of run-time speed and the amount of memory they consume

Compilers that apply code-improving transformations are called Optimizing Compilers

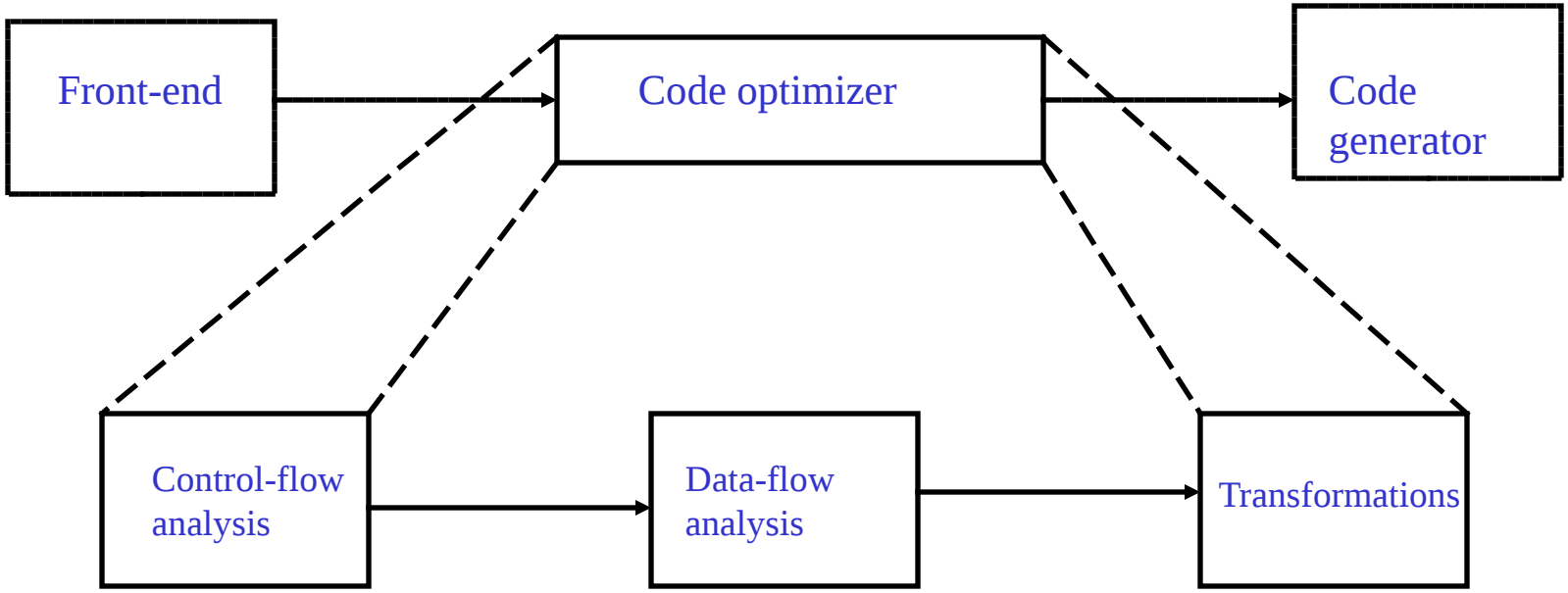
Machine Independent improvements address the logic of the program

Machine Dependent improvements utilize special features of the target instruction set, including registers and special addressing modes

Optimization Components



Organization of a code optimizer



Flow Analysis

Loops represent the most computationally intensive part of a program. Improvements to loops will produce the most significant effect

Local Optimizations are performed on basic blocks of code

Global Optimizations are performed on the whole code

A Basic Block is a sequence of instructions that is only entered at the start and exited at the end, with no jumps into or out of the middle.

That is, a basic block begins at a procedure or the target of a jump

A basic block ends at the start of the next basic block or the end of the program

Criteria for code-improvement Transformations

1. Transformations must preserve the meaning of programs

2. A transformation must, on the average, speed up programs by a measurable amount

3. A transformation must be worth the effort

Function Preserving Transformations

1. Common subexpression eliminations
2. Copy propagations
3. Dead and unreachable code elimination
4. Constant Folding

Example: C code

```
void quicksort(m, n)
int m, n;
{
  int I, j;
  if (n <= m ) return;
  /* fragment begins here */
  i = m-1; j = n; v = a[n];
  while(1)   {
     do i = i+1; while( a[i] < v );
        do j = j-1; while( a[j] > v );
        if( i >= j ) break;
        x = a[i]; a[i] = a[j];     a[j] = x;
     }
  x = a[i]; a[i] = a[n]; a[n]= x;
  /* fragment ends here */
  quicksort(m, j); quicksort(i+1, n);
}
```

Augmented 3AC

An augmented 3 address code language to simplify the code...

Let *a* be an array of integers starting at byte address a_0

`a[add]` on the left-hand-side of an assignment is the address a_0**`+add`**

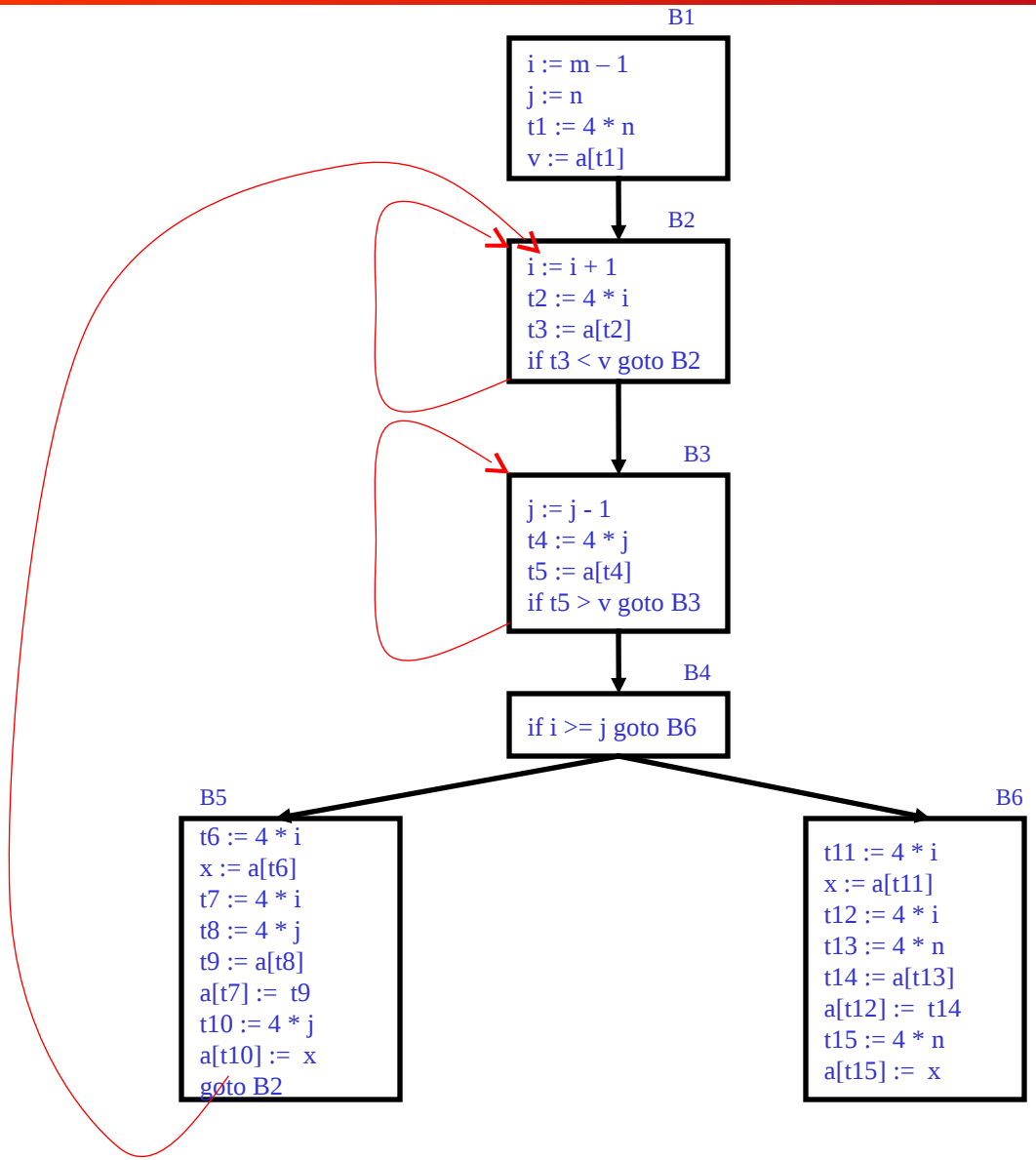
`a[add]` on the right-hand-side of an assignment is the value of the element of the array at address a_0**`+add`**

Since integers are stored in 4 bytes the offset address of an element *a[i]* is **`4*i`**

Example: 3AC

```
01) i := m - 1
02) j := n
03) t1 := 4 * n
04) v := a[t1]
05) i := i + 1
06) t2 := 4 * i
07) t3 := a[t2]
08) if t3 < v goto 5
09) j := j - 1
10) t4 := 4 * j
11) t5 := a[t4]
12) if t5 > v goto 9
13) if i >= j goto 23
14) t6 := 4 * i
15) x := a[t6]
16) t7 := 4 * i
17) t8 := 4 * j
18) t9 := a[t8]
19) a[t7] := t9
20) t10 := 4 * j
21) a[t10] := x
22) goto 5
23) t11 := 4 * i
24) x := a[t11]
25) t12 := 4 * i
26) t13 := 4 * n
27) t14 := a[t13]
28) a[t12] := t14
29) t15 := 4 * n
30) a[t15] := x
```

Basic Blocks



Local Optimizations

Block5 before

```
t6 := 4 * i
x := a[t6]
t7 := 4 * i
t8 := 4 * j
t9 := a[t8]
a[t7] := t9
t10 := 4 * j
a[t10] := x
goto Block2
```

Block5 after

```
t6 := 4 * i
x := a[t6]
t8 := 4 * j
t9 := a[t8]
a[t6] := t9
a[t8] := x
goto Block2
```

redundant calculations removed

Local Optimizations

Block6 before

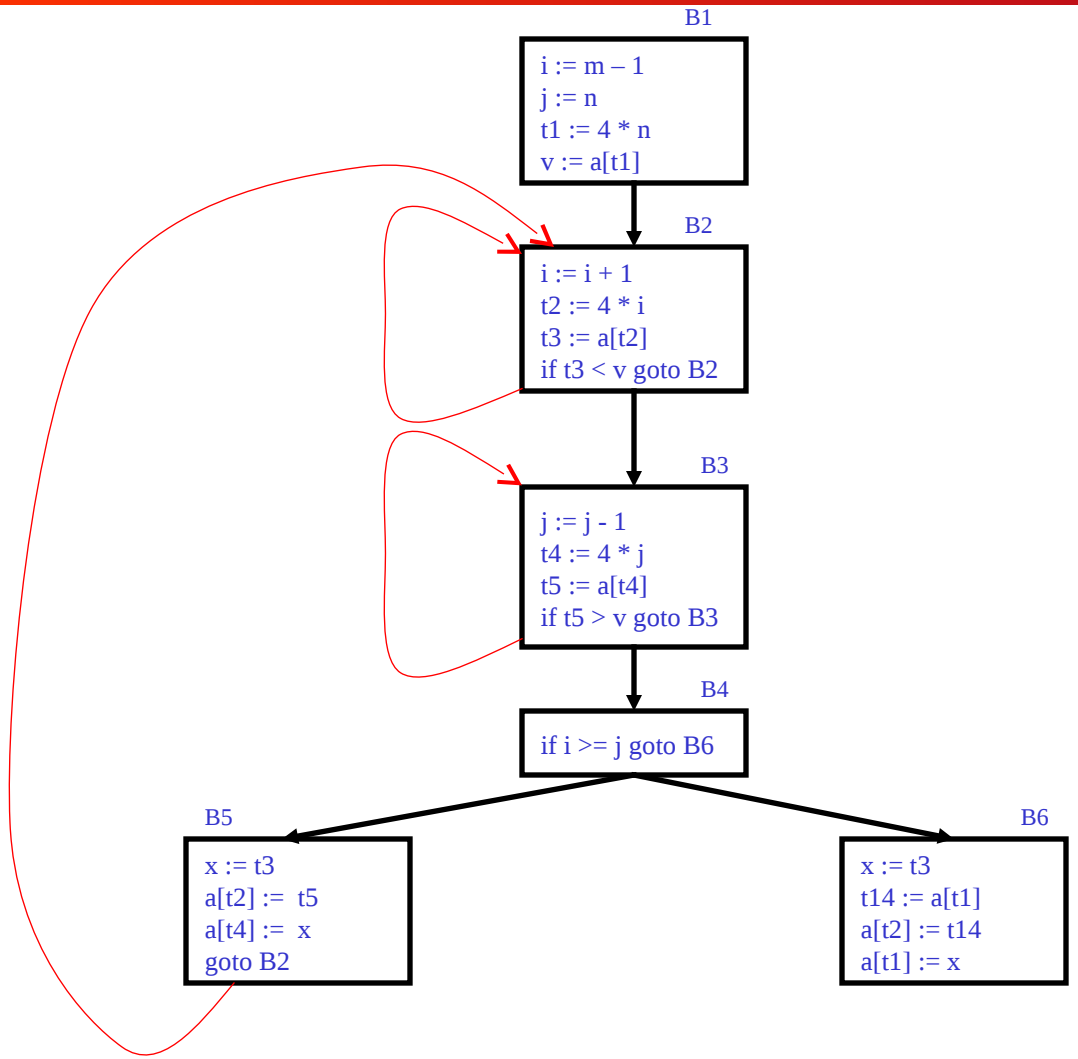
```
t11 := 4 * i
x := a[t11]
t12 := 4 * i
t13 := 4 * n
t14 := a[t13]
a[t12] := t14
t15 := 4 * n
a[t15] := x
```

Block6 after

```
    t11 := 4 * i
x := a[t11]
    t13 := 4 * n
    t14 := a[t13]
    a[t11] := t14
    a[t13] := x
```

redundant calculations removed

Global Optimizations



After removing redundant calculations over all blocks

Loop Optimization

1. Code Motion
2. Reduction in Strength
3. Induction Variables elimination

Code Motion

Code Motion decreases the amount of code in a loop

Before

```
while ( i <= limit + 2) /*statement does not change limit */
```

After

```
t = limit + 2
while ( i <= t) /*statement does not change limit or t*/
```

Reduction in Strength

In Block2 whenever i increases by 1, t2 increases by 4
In Block3 whenever j decreases by 1, t4 decreases by 4
Addition and Subtraction can be used instead of the more computationally expensive multiplication (t2 and t4 must be initialized).

Before

```
i := m - 1
j := n
t1 := 4 * n
v := a[t1]

Block2:
i := i + 1
t2 := 4 * i
t3 := a[t2]
if t3 < v goto B2
Block3:
j := j - 1
t4 := 4 * j
t5 := a[t4]
```

After

```
i := m - 1
j := n
t1 := 4 * n
v := a[t1]
t2 := 4 * i
t4 := 4 * j
Block2:
i := i + 1
        t2 := t2 + 4
t3 := a[t2]
if t3 < v goto B2
Block3:
j := j - 1
t4 := t4 - 4
t5 := a[t4]
```

Induction Variables elimination

In Block2 whenever i increases by 1, t2 increases by 4,
i and t2 are called induction variables.

In Block3 whenever j decreases by 1, t4 decreases by 4,
j and t4 are induction variables, too.

If there are two or more induction variables in a loop, it may be possible to get rid of all but one

Before

Block4:

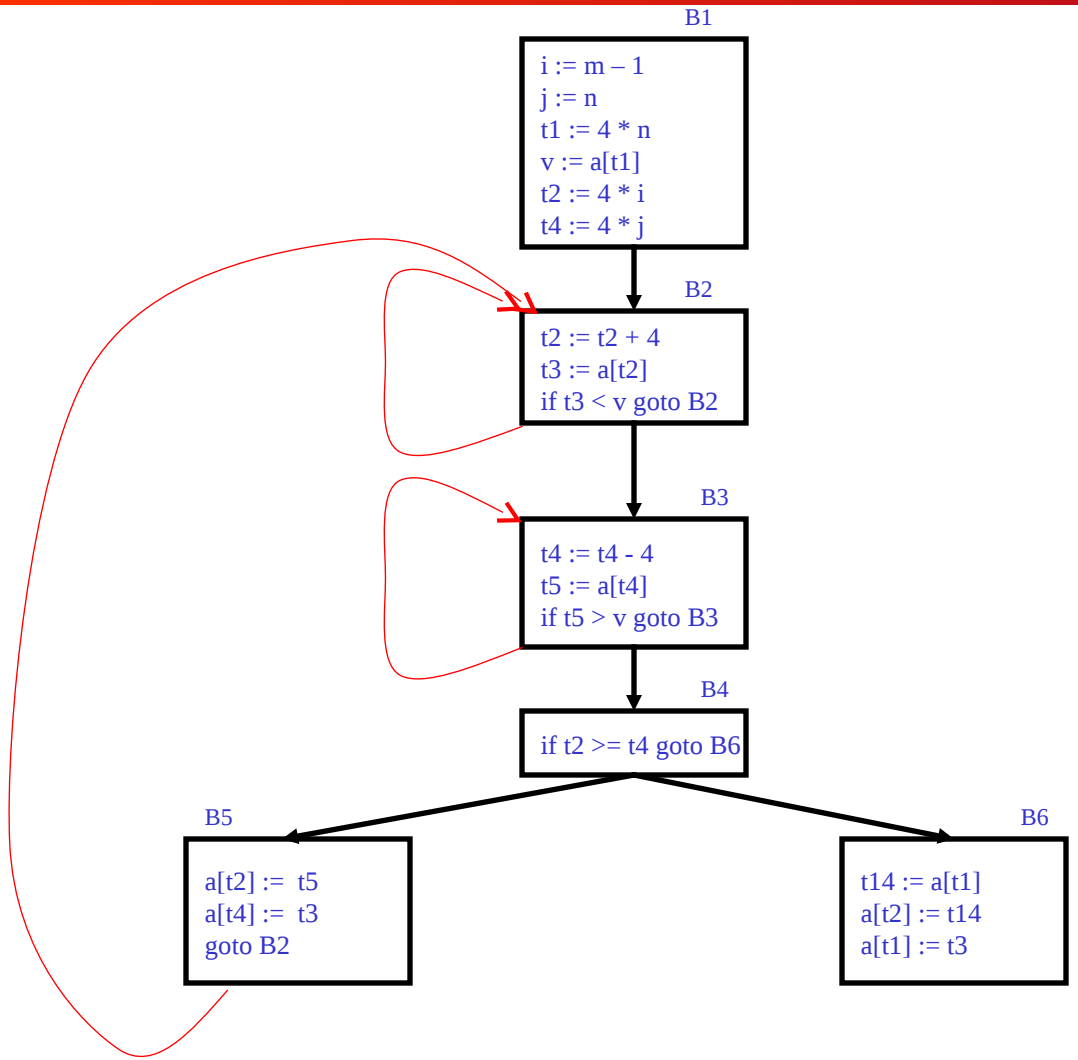
if i >= j goto B6

After

Block4:

if t2 >= t4 goto B6

Loop Optimization



After reduction in strength and induction-variable elimination

Peephole Optimization

Removing Redundant Load and Stores

3AC

```
a := b + c;
d := a - e;
```

Machine Code

```
mov registera b
add registera c
mov a registera        -- redundant if
mov registera a        -- a is no longer used
sub registera e
mov d registera
```

Others

Algebraic Simplification

x := x + 0
x := x * 1

Use of Registers to store the most used variables because access time is much quicker than memory

Use of specialized instructions

mov a
add registera 1
mov a registera

could be just inc a

Using shift to left instead of multiplication by powers of 2
Using shift to right instead of division into powers of 2