Machine-Independent Optimizations

Kavitha K R

Department of Computer Science

Code Optimization Methods

- There are a number of ways in which a compiler can improve a program without changing the function it computes.
  - Common-subexpression elimination
  - Copy propagation
  - Dead-code elimination
  - Constant folding
  - Code motion
  - Induction-variable elimination

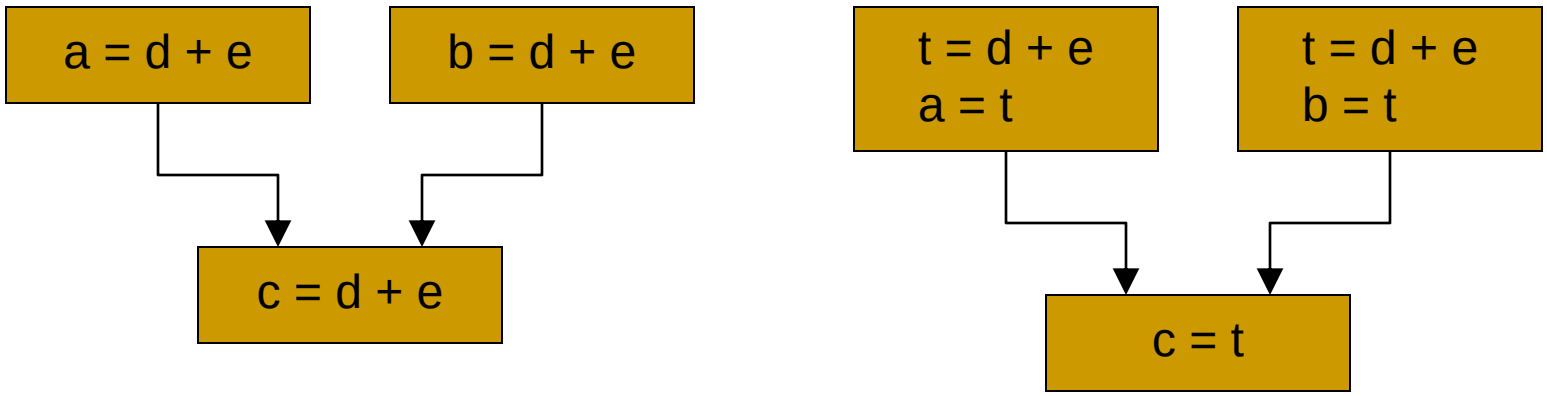
Common-Subexpression Elimination

- An occurrence of an expression E is called a common subexpression if E was previously computed and the values of the variables in E have not changed since the previous computation.

  Avoid recomputing E if can be used its previously computed value; that is, the variable x to which the previous computation of E was assigned has not changed in the interim.

Copy Propagation (1)

- This optimization concerns assignments of the form u = v called copy statements.
- The idea behind the copy-propagation transformation is to use v for u, wherever possible after the copy statement u = v.
- Copy propagation work example:



Copy Propagation (2)

- The assignment x = t3 in block B5 is a copy.
  Here is the result of copy propagation applied to B5.

- This change may not appear to be an improvement, but it gives the opportunity to eliminate the assignment to x.
- One advantage of copy propagation is that it often turns the copy statement into dead code.

Dead-code Elimination

- Code that is unreachable or that does not affect the program (e.g. dead stores) can be eliminated.

  While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.
- Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

Dead-code Elimination: Example

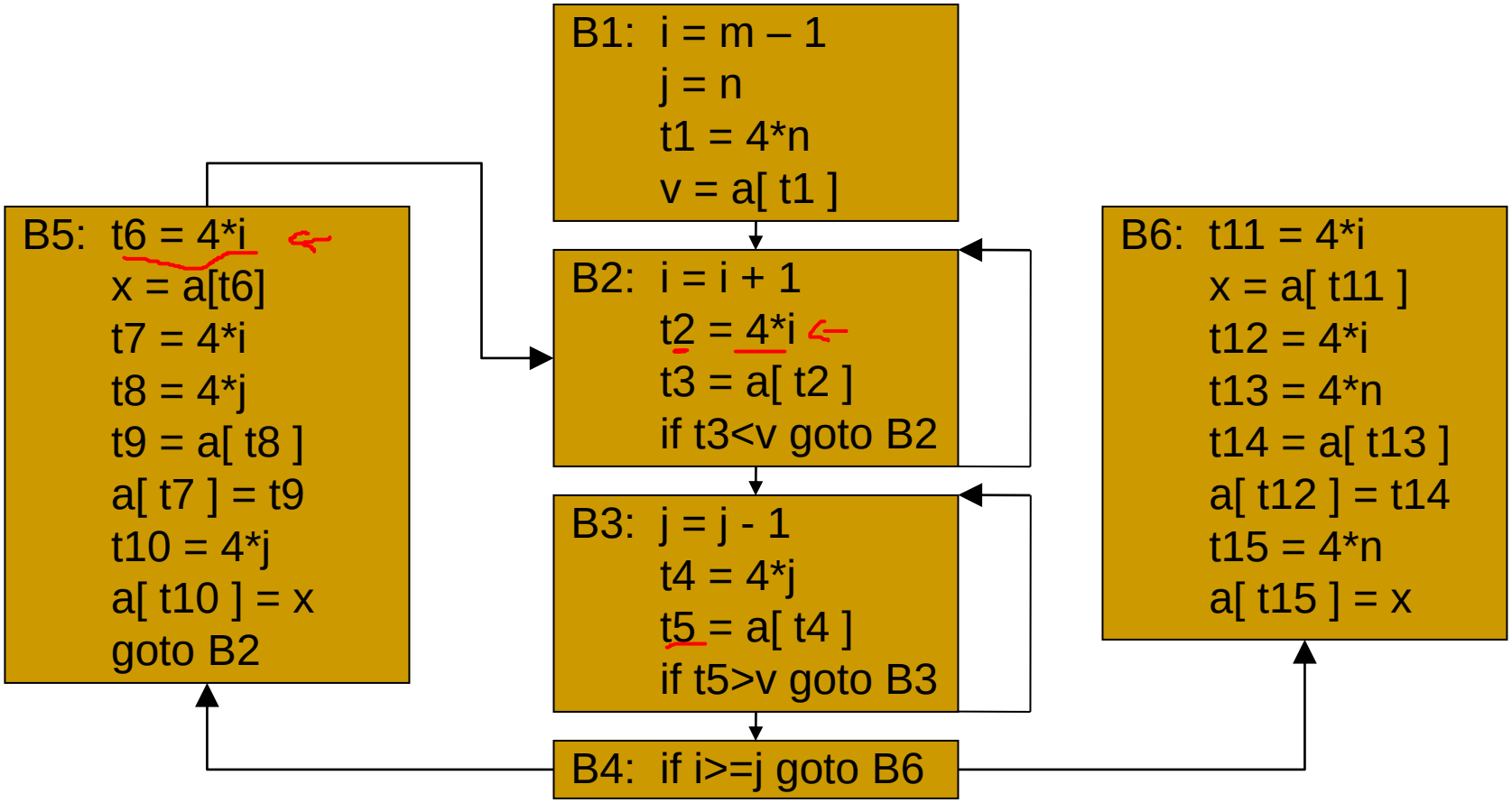
- In the example below, the value assigned to i is never used, and the dead store can be eliminated. The first assignment to global is dead, and the third assignment to global is unreachable; both can be eliminated.

```
int global;
void f ()
{
    int i;
    i = 1; /* dead store */
    global = 1; /* dead store */
    global = 2;
    return;
    global = 3; /* unreachable */
}
```

```
int global;
void f ()
{
        global = 2;
        return;
}
```

Before and After Dead Code Elimination

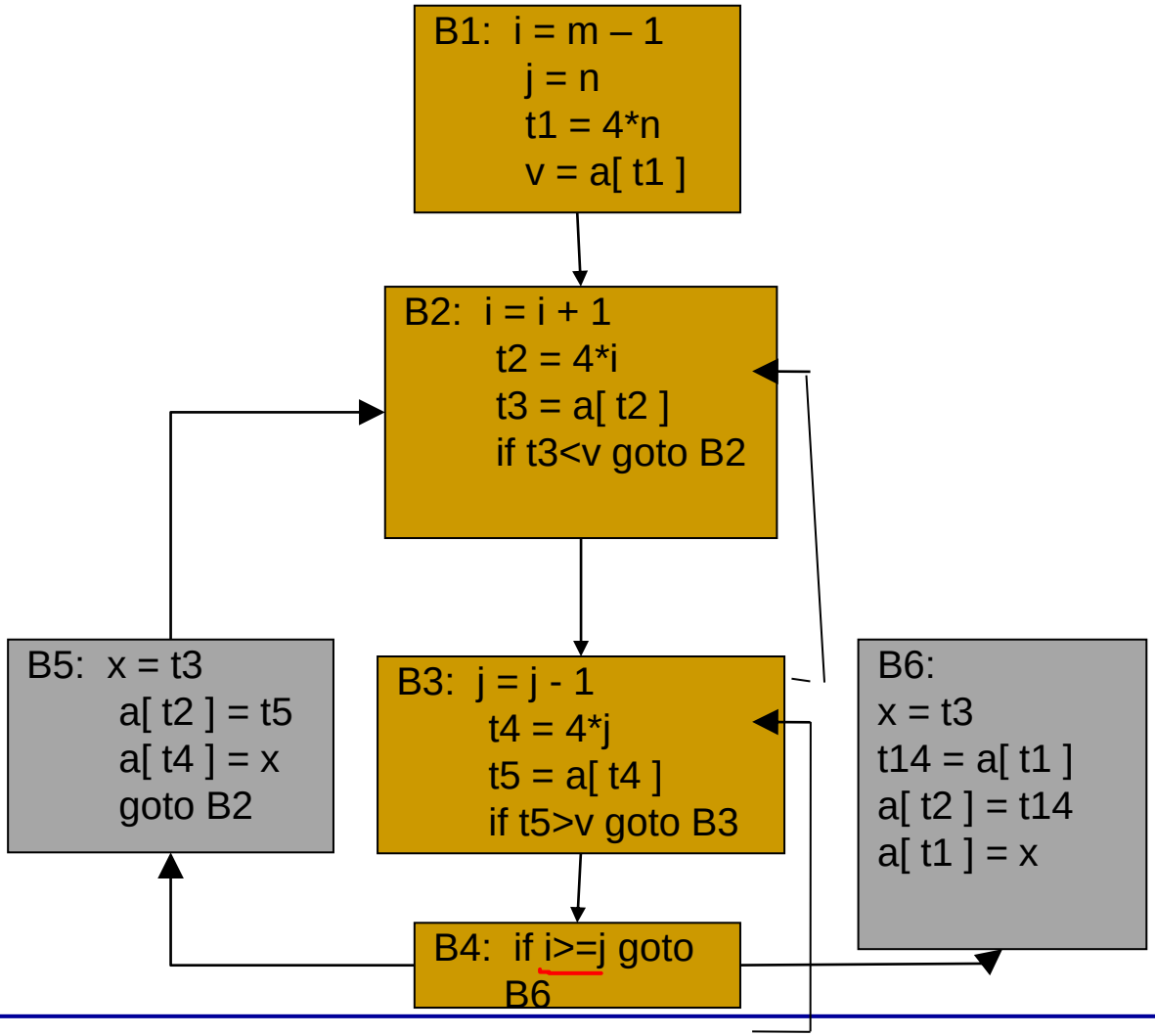
Quick Sort Example



```
B5:  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 B2
```

```
B6:  t11 = 4*i
     x = a[ t11 ]
     t12 = 4*i
     t13 = 4*n
     t14 = a[ t13 ]
     a[ t12 ] = t14
     t15 = 4*n
     a[ t15 ] = x
```

Flow Graph After optimization



Code Motion

- Code motion decreases the amount of code in a loop.

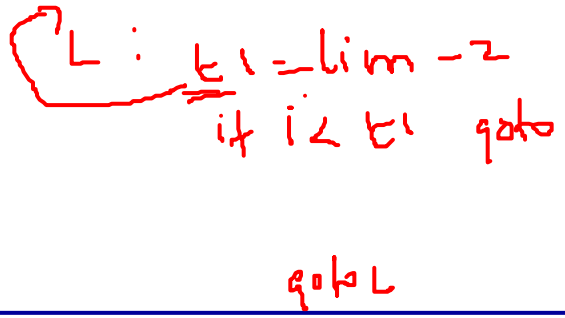
  This transformation takes an expression that yields the same result independent of the number of times a loop is executed (**a loop-invariant computation**) and evaluates the expression before the loop.

  Evaluation of **limit - 2** is a loop-invariant computation in the following while-statement :

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

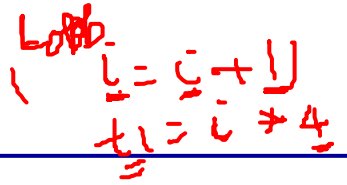
- Code motion will result in the equivalent code:

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

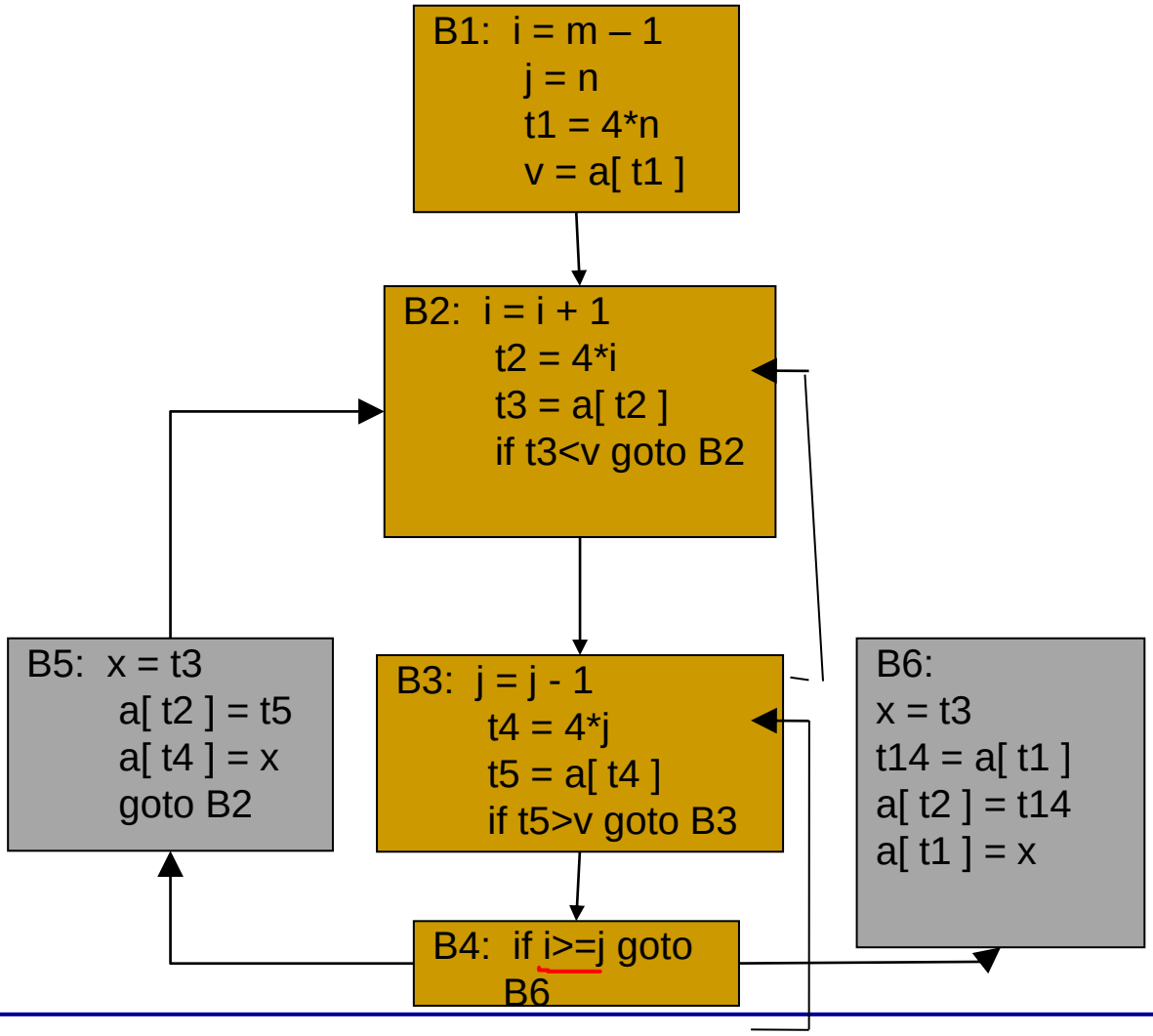


Induction-Variable (IV) Elimination

- Variable x is said to be an "induction variable“ if there is a positive or negative constant c such that each time x is assigned, its value increases by c.
- For instance, i and t2 are induction variables in the loop containing B2 of QuickSort example.
- Induction variables can be computed with a single increment (addition or subtraction) per loop iteration. The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as strength reduction.



Flow Graph before IV optimization



Flow Graph After IV Elimination