CS536

Machine-Independent Optimizations

A Sahu

CSE, IIT Guwahati

Outline

- Theme of II part of course
  - **Machine Independent Code Optimization (Ch9)**
  - **Cache and Locality Aware Compiler Optimization (Ch10)**
  - **Parallelism Aware Compiler Optimization (Ch11)**
- Machine Independent Optimization
- Basic Rule for Programming : Code Optimization (loop), High level
- Standard Optimizations

Basic Rule for Programming : Code Optimization (loop), High level

Given Base Code

```
void combine2(vec_ptr v, int *dest){
 int i;
 *dest = 0;
 for (i = 0; i < vec_length(v); i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }

}
```

Reduction in Strength/Loop Inv Code

```
void combine2(vec_ptr v, int *dest){
  int i;
 *dest = 0;
 int length = vec_length(v);
 for (i = 0; i < length; i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }

}
```

Reduction in Strength

```
void combine2(vec_ptr v, int *dest){
   int i;
 *dest = 0;
 int length = vec_length(v);

 for (i = 0; i < length; i++) {
     int val;
     get_vec_element(v, i, &val);
     *dest += val;
   }

}
```

Reduction in Strength

```
void combine3(vec_ptr v, int *dest){

   int i;
   *dest = 0;
   int length = vec_length(v);
   int *data = get_vec_start(v);

   for (i = 0; i < length; i++) {
      *dest += data[i];
   }

}
```

Eliminate Unneeded Memory Refs

```
void combine4(vec_ptr v, int *dest){

  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int sum = 0;

  for (i = 0; i < length; i++)
    sum += data[i];

  *dest = sum;

}
```

Use Pointer Code

- Use pointers rather than array references
- *Warning*: Some compilers do better job optimizing array code → **We want to do in compiler**

```
void combine4p(vec_ptr v, int *dest){
   int length = vec_length(v);
   int *data = get_vec_start(v);
   int *dend = data+length;
   int sum = 0;
   while (data < dend) {
      sum += *data; data++;
   }
   *dest = sum;
}
```

Machine Independent Code Optimization

Causes of Redundancy

- Redundancy is available at the source level
  - Due to recalculations while one calculation is necessary.
- Redundancies in address calculations
  - Redundancy is a side effect of having written the program in a high-level language
  - where referrals to elements of an array or fields in a structure is done through accesses like A[i][j] or X -> f1.

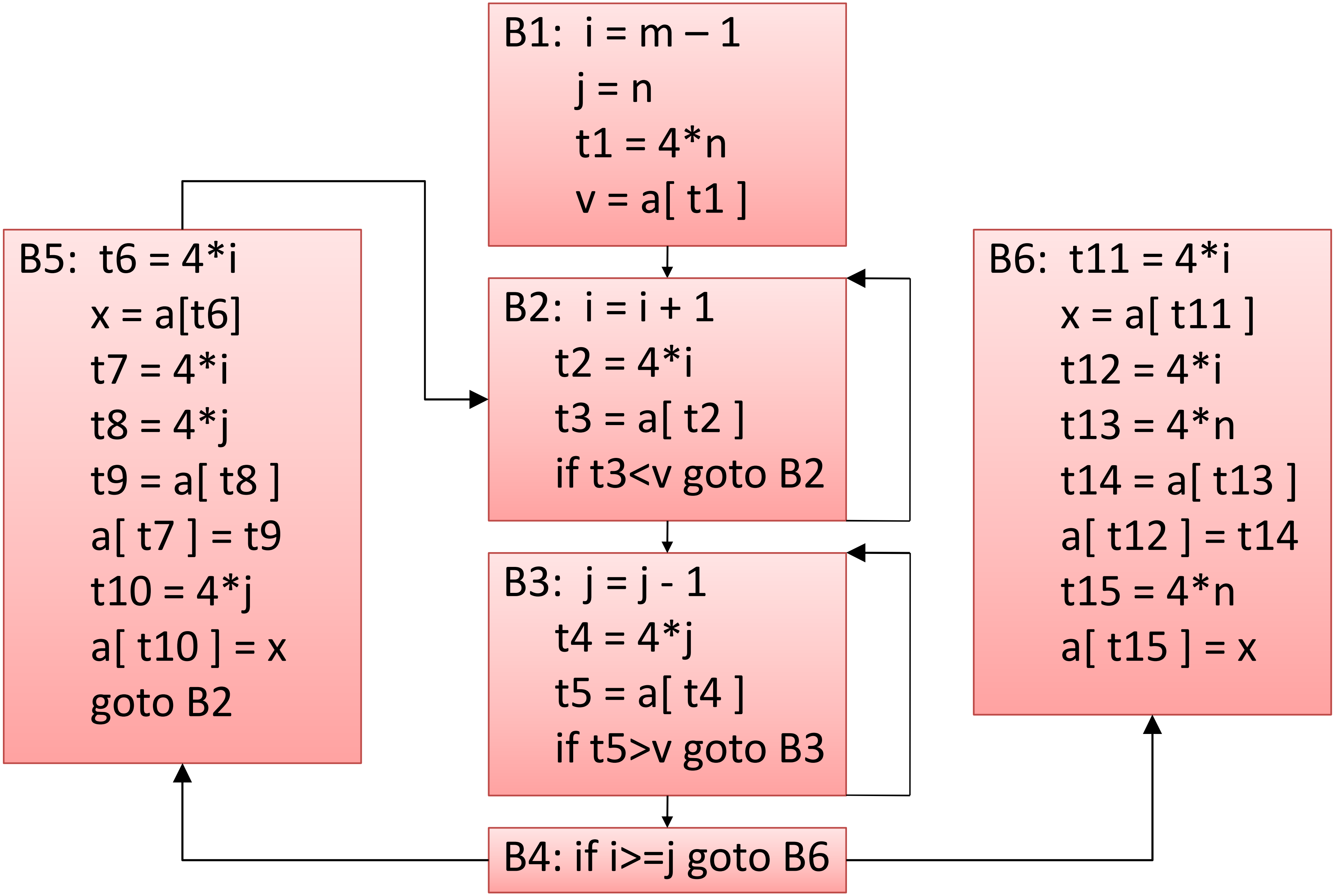
Causes of Redundancy

- As a program is compiled,
- Each of high-level data-structure accesses
  - array access and structure access
- Get expands into a number of low-level arithmetic operations
  - Such as the computation of the location of the [i, j]-th element of a matrix A.
- Accesses to the same data structure often share many common low-level operations.

A Running Example: Quicksort

```
void quicksort (int m, int n) {
  /* recursively sorts a[ m ] through a[ n ] */
        int i , j, v, x;
        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;  /* swap a[ i ] , a[ n ] */
        /* fragment ends here */
        quicksort ( m, j ); quicksort ( i + 1 ,n );
  }
```

Flow Graph for Quicksort Fragment



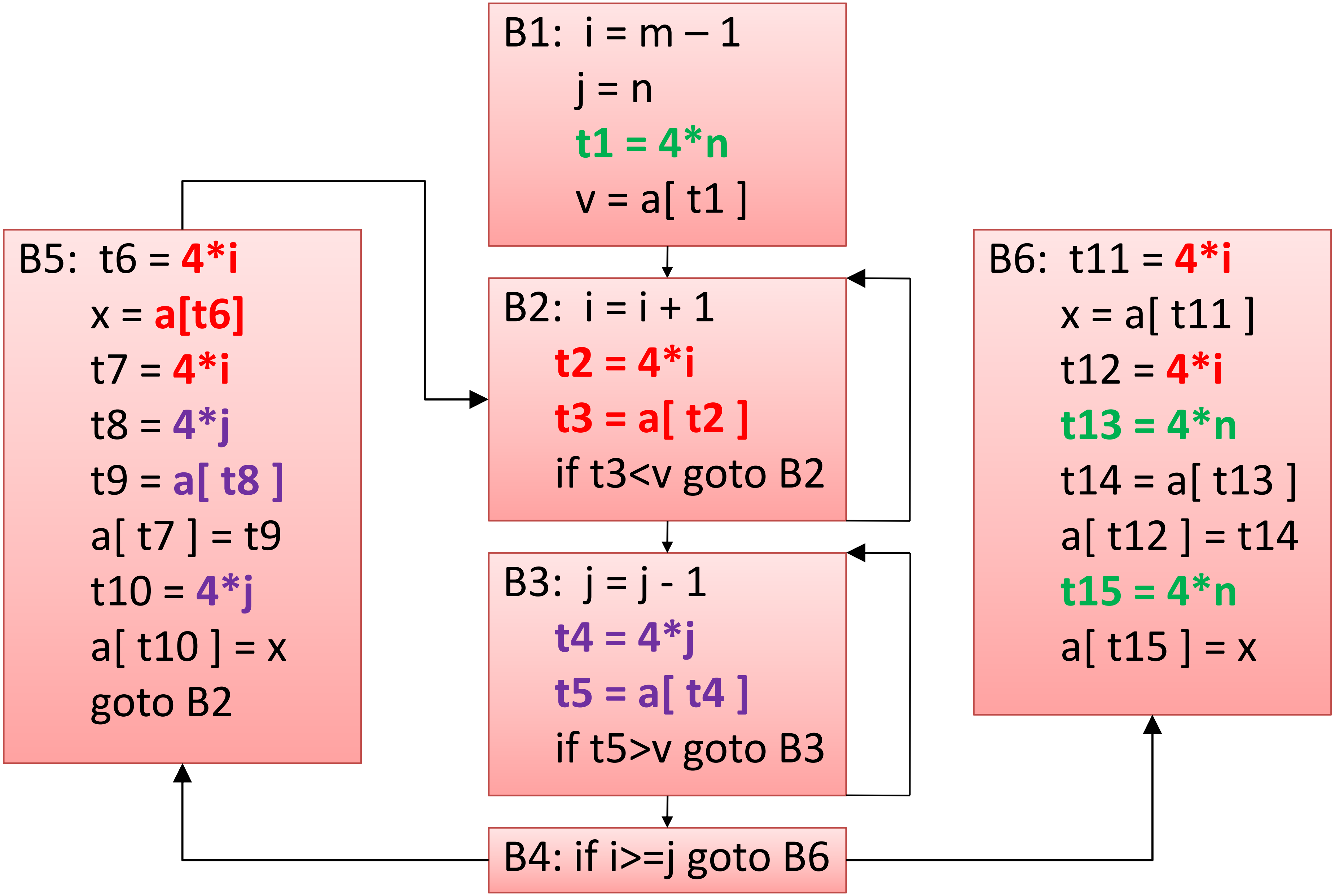
Semantics-Preserving Transformations

- 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

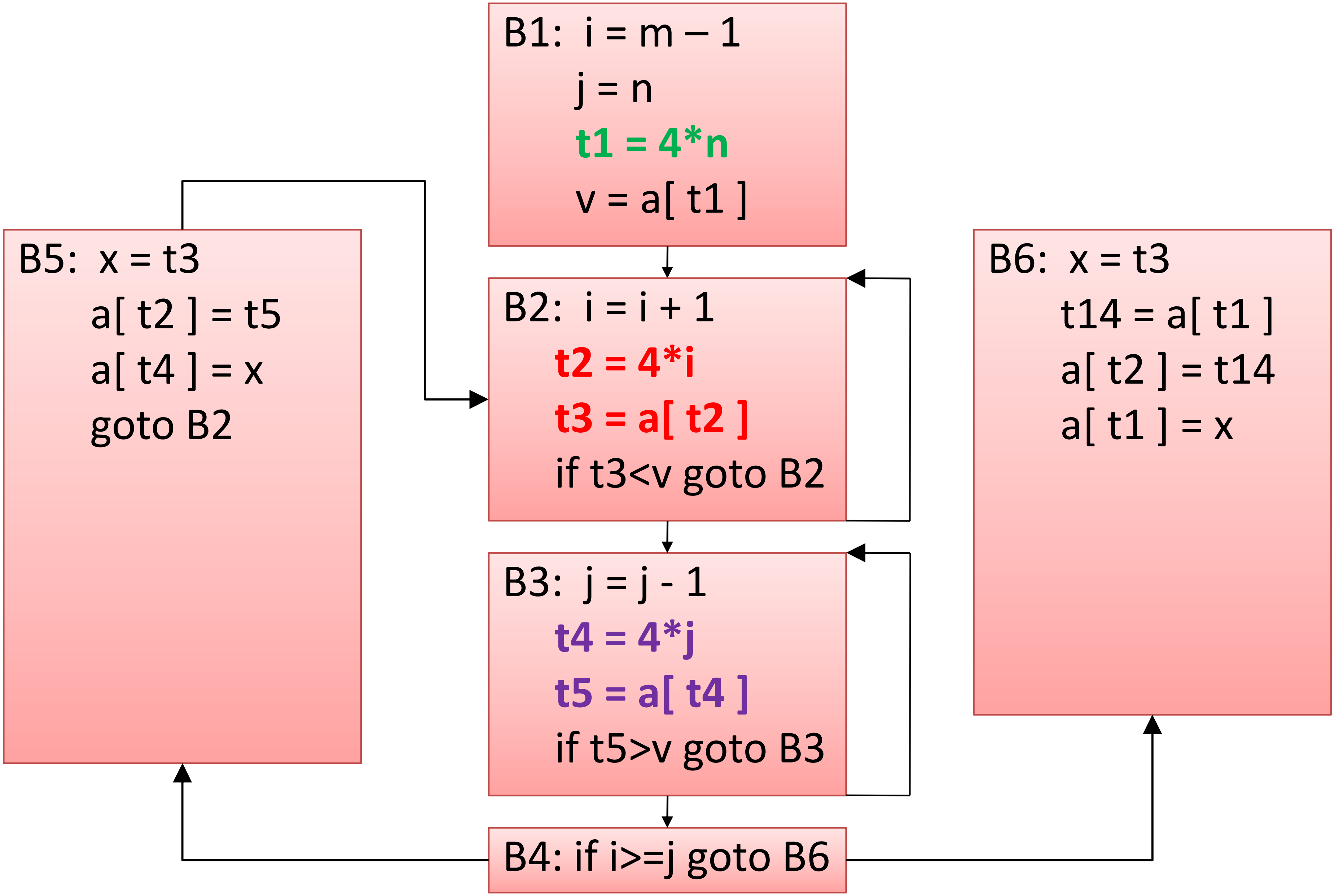
Common-Subexpression Elimination

- 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.

Common Sub Expr. Elimination

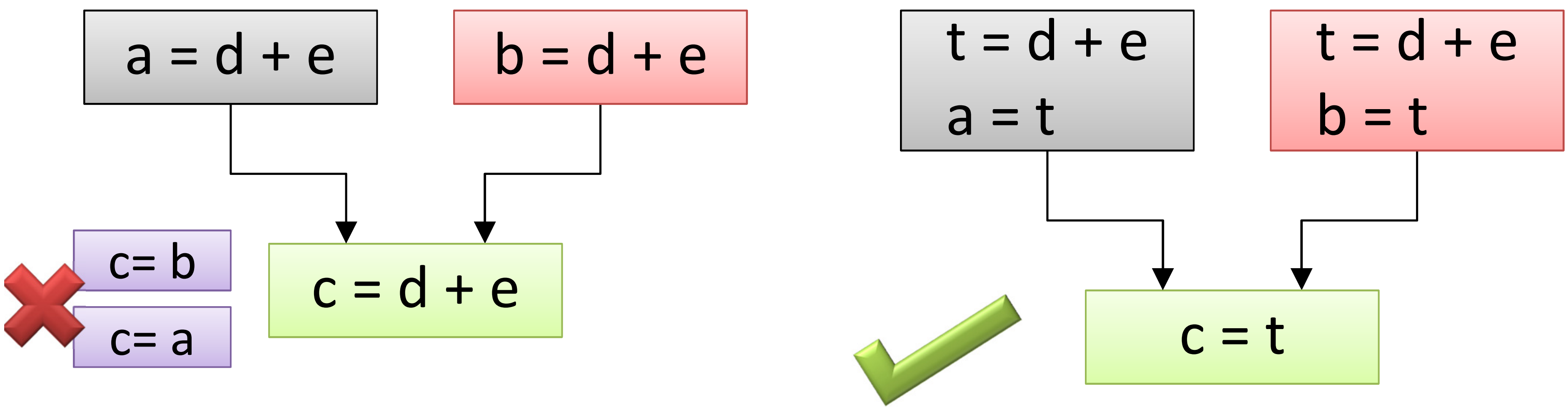


Flow Graph After C.S. Elimination



Copy Propagation

- 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

- The assignment x = t3 in block B5 is a copy.

  Here is the result of copy propagation applied to B5.

```
B5:  x = t3
     a[ t2 ] = t5
     a[ t4 ] = x
     goto B2
```

```
B5:  x = t3
     a[ t2 ] = t5
     a[ t4 ] = t3
     goto B2
```

- **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.**