CS738: Advanced Compiler Optimizations

Welcome & Introduction

Amey Karkare

karkare@cse.iitk.ac.in

http://www.cse.iitk.ac.in/~karkare/cs738

Department of CSE, IIT Kanpur

About the Course

- Program Analysis

[1] “Democracy is the government of the people, by the people, for the people” - Abraham Lincoln

About the Course

- Program Analysis
- Analysis of a Program, by a Program, for a Program[1]

[1] "Democracy is the government of the people, by the people, for the people" - Abraham Lincoln

About the Course

- Program Analysis
- Analysis of a Program, by a Program, for a Program[1]
  - Of a Program – User Program

[1] "Democracy is the government of the people, by the people, for the people" - Abraham Lincoln

About the Course

- Program Analysis
- Analysis of a Program, by a Program, for a Program[1]
  - Of a Program – User Program
  - By a Program – Analyzer (Compiler, Runtime)

[1] “Democracy is the government of the people, by the people, for the people” - Abraham Lincoln

About the Course

- Program Analysis
- Analysis of a Program, by a Program, for a Program[1]
  - Of a Program – User Program
  - By a Program – Analyzer (Compiler, Runtime)
  - For a Program – Optimizer, Verifier

[1] “Democracy is the government of the people, by the people, for the people” - Abraham Lincoln

About the Course

- Program Analysis
- Analysis of a Program, by a Program, for a Program[1]
  - Of a Program – User Program
  - By a Program – Analyzer (Compiler, Runtime)
  - For a Program – Optimizer, Verifier
- Transforming user program based on the results of the analysis

[1] "Democracy is the government of the people, by the people, for the people" - Abraham Lincoln

Expectations from You

- Basic Compiler Knowledge

Expectations from You

- Basic Compiler Knowledge
- Write Code

Expectations from You

- Basic Compiler Knowledge
- Write Code
- Willingness to understand and modify large code bases

Expectations from You

- Basic Compiler Knowledge
- Write Code
- Willingness to understand and modify large code bases
- Read and present state-of-the-art reseach papers

Your Expectations

? Share through the Google Form

Quick Quizzes (QQs)

- There will be small quizzes (10-15 min duration) during the class.

Quick Quizzes (QQs)

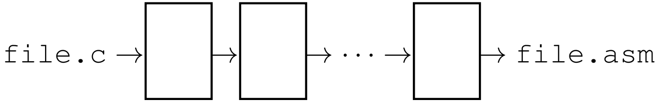
- There will be small quizzes (10-15 min duration) during the class.
- These can be announced or un-announced (surprize quizzes).

Quick Quizzes (QQs)

- There will be small quizzes (10-15 min duration) during the class.
- These can be announced or un-announced (surprize quizzes).
- Always bring a pen and some loose papers to the class

QQ #1 (Ungraded)

- What are the vaious phases of a typical compiler? (5 minutes)



Assignments

- Short assignments to apply the lecture material.

Assignments

- Short assignments to apply the lecture material.
- Assignments will have some written and some programming tasks.

Assignments

- Short assignments to apply the lecture material.
- Assignments will have some written and some programming tasks.
- 4–5 Assignments for the semester

Using Program Analysis

- Compiler Code Optimizations

Using Program Analysis

- Compiler Code Optimizations
- Why are optimizations important?

Using Program Analysis

- Compiler Code Optimizations
- Why are optimizations important?
- Why not write optimized code to begin with?

Using Program Analysis

- Compiler Code Optimizations
- Why are optimizations important?
- Why not write optimized code to begin with?
- Where do optimizations fit in the compiler flow?

Code Optimization

- Machine Independent

Code Optimization

- Machine Independent
  - Remove redundancy introduced by the Programmer

Code Optimization

- Machine Independent
  - Remove redundancy introduced by the Programmer
  - Remove redundancy not required by later phases of compiler

Code Optimization

- Machine Independent
  - Remove redundancy introduced by the Programmer
  - Remove redundancy not required by later phases of compiler
  - Take advantage of algebraic properties of operators

Code Optimization

- Machine Independent
  - Remove redundancy introduced by the Programmer
  - Remove redundancy not required by later phases of compiler
  - Take advantage of algebraic properties of operators
- Machine dependent

Code Optimization

- Machine Independent
  - Remove redundancy introduced by the Programmer
  - Remove redundancy not required by later phases of compiler
  - Take advantage of algebraic properties of operators
- Machine dependent
  - Take advantage of the properties of target machine

Code Optimization

- Machine Independent
  - Remove redundancy introduced by the Programmer
  - Remove redundancy not required by later phases of compiler
  - Take advantage of algebraic properties of operators
- Machine dependent
  - Take advantage of the properties of target machine
- Optimization must preserve the semantics of the original program!

Machine Independent Optimizations

Motivational Example

```
void quicksort(int m, int n)
/* recursively sort a[m] through a[n] */
{
    int i, j;
    int v, x;
    if(n <= m) return;
    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;
    quicksort(m,j); quicksort(i+1,n);
}
```

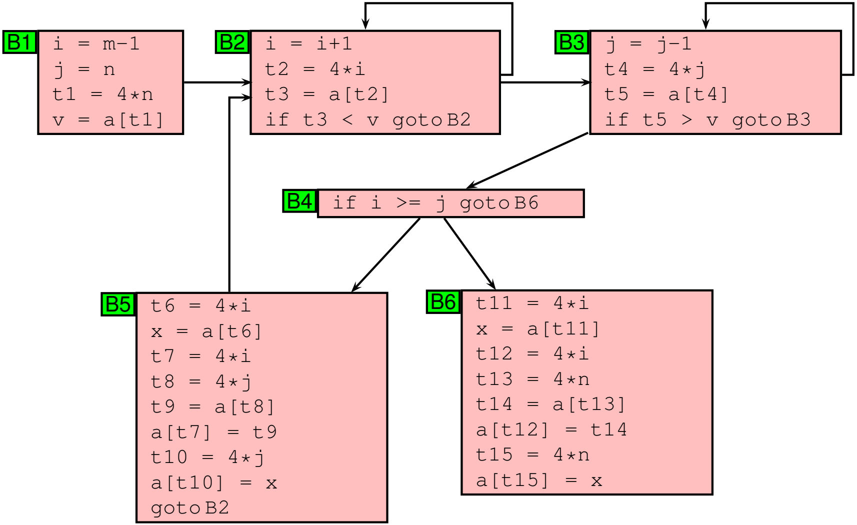
Motivational Example

```
void quicksort(int m, int n)
/* recursively sort a[m] through a[n] */
{
    int i, j;
    int v, x;
    if(n <= m) return;
    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;
    quicksort(m,j); quicksort(i+1,n);
}
```

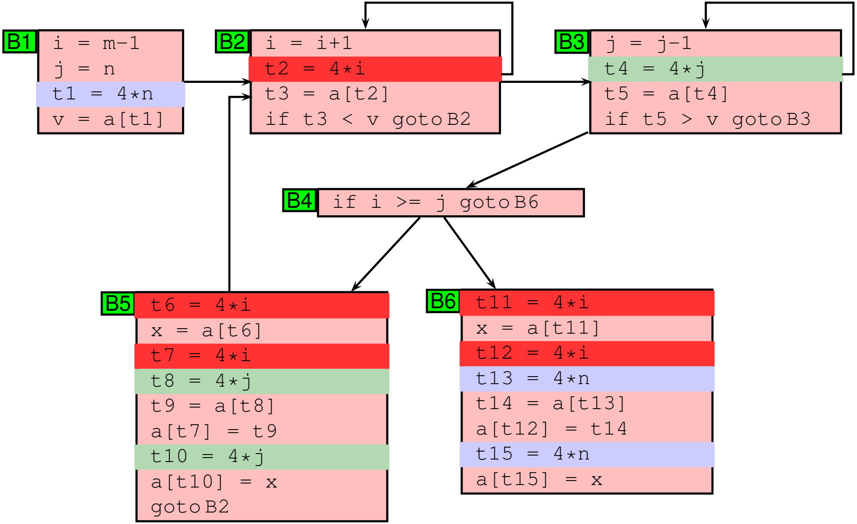
```
( 1) i = m-1
( 2) j = n
( 3) t1 = 4*n
( 4) v = a[t1]
( 5) i = i+1
( 6) t2 = 4*i
( 7) t3 = a[t2]
( 8) if t3 < v goto (5)
( 9) j = j-1
(10) t4 = 4*j
(11) t5 = a[t4]
(12) if t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 = 4*i
(15) x = a[t6]
(16) t7 = 4*i
(17) t8 = 4*j
(18) t9 = a[t8]
(19) a[t7] = 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
```

```
( 1)  i = m-1
( 2)  j = n
( 3)  t1 = 4*n
( 4)  v = a[t1]
( 5)  i = i+1
( 6)  t2 = 4*i
( 7)  t3 = a[t2]
( 8)  if t3 < v goto (5)
( 9)  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
```

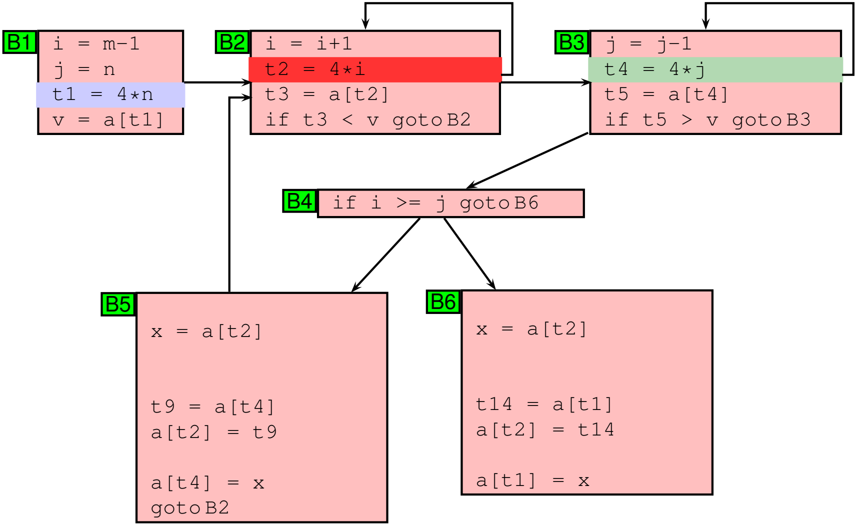
Common Subexpresion Elimination



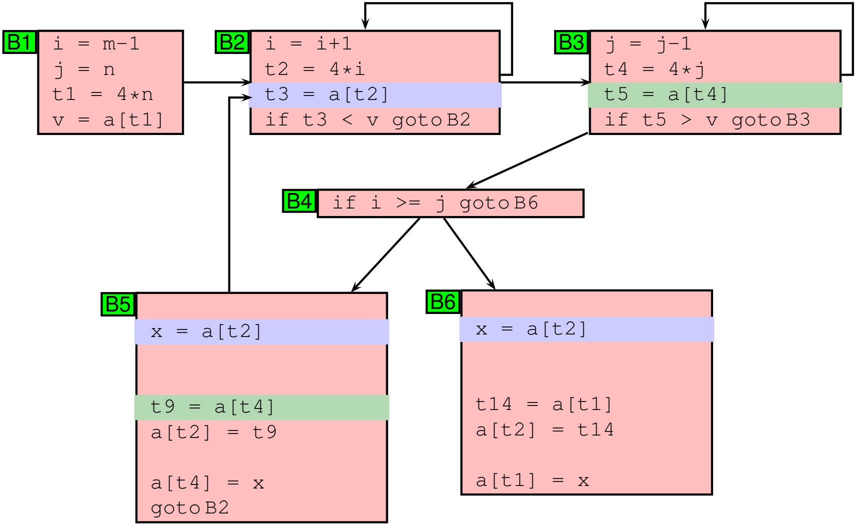
Common Subexpresion Elimination



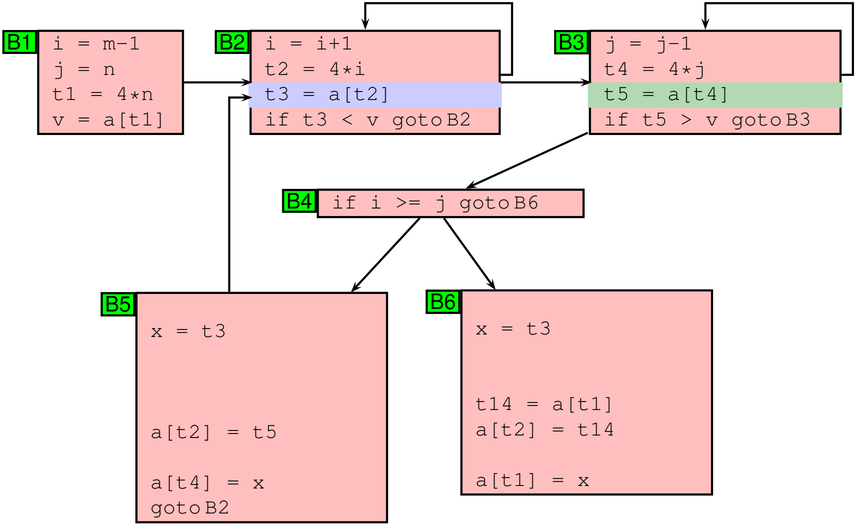
Common Subexpresion Elimination



Common Subexpresion Elimination

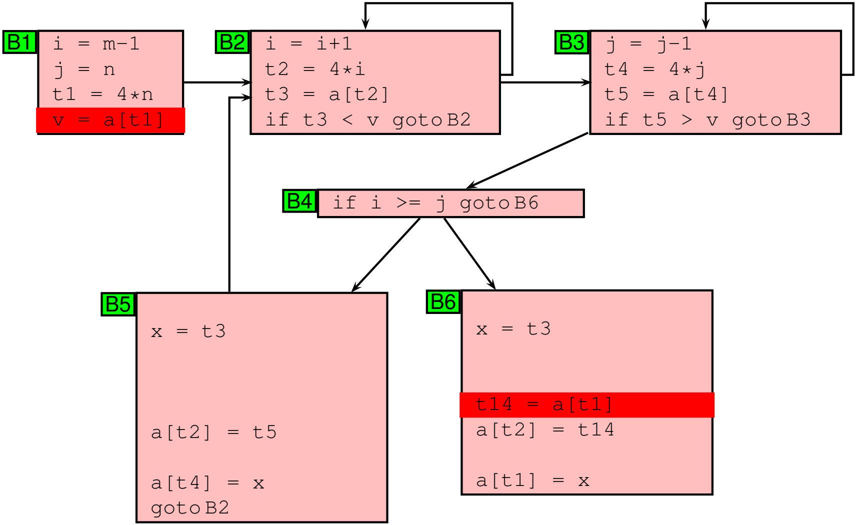


Common Subexpresion Elimination



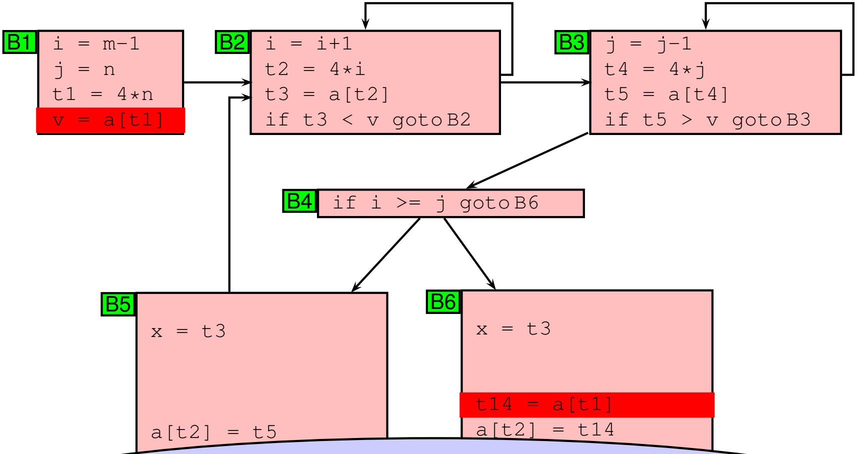
Common Subexpresion Elimination

Did we miss one expression?



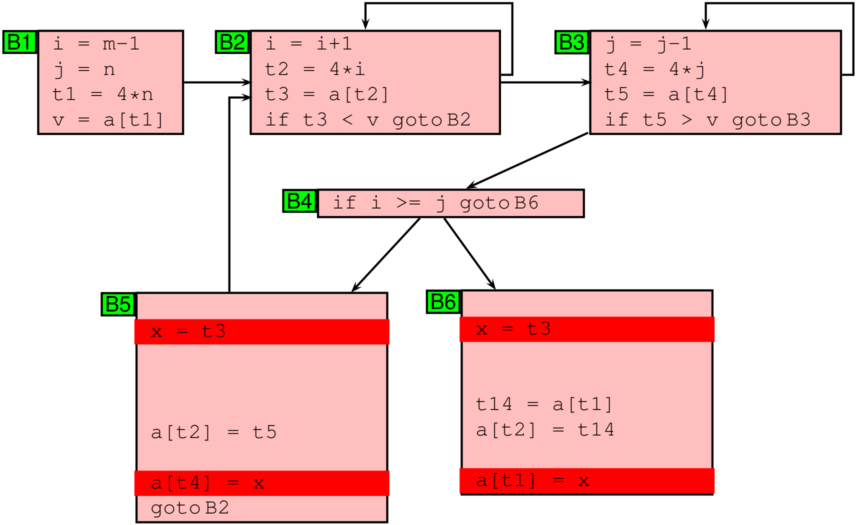
Common Subexpresion Elimination

Did we miss one expression?

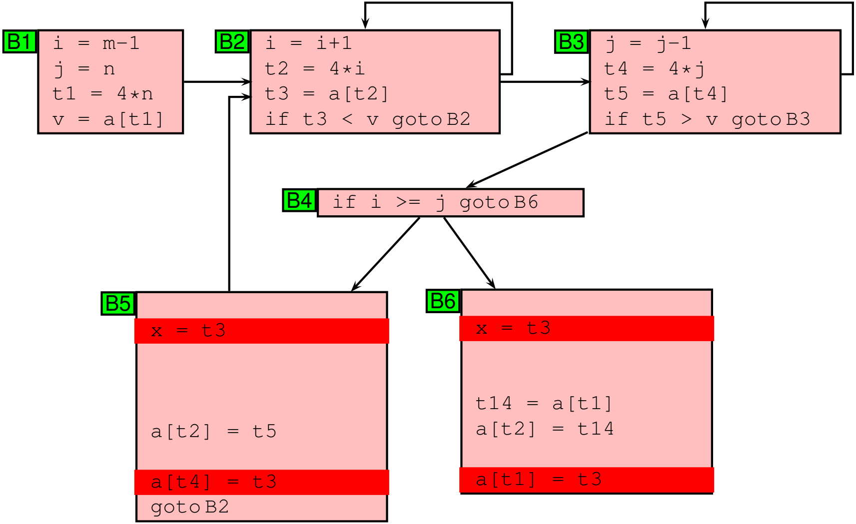


Elimination not safe as a[] is modified on path
B1→B2→B3→B4→B5→B2→B3→B4→B6

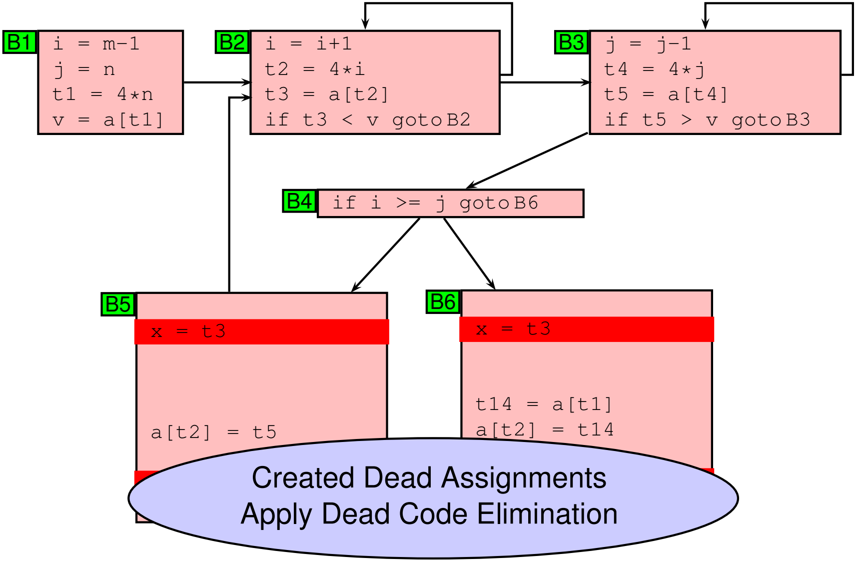
Copy Propagation



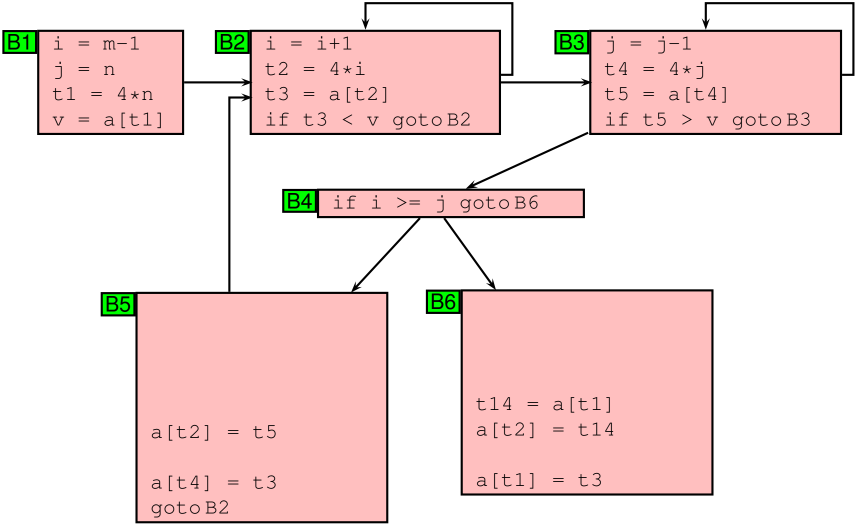
Copy Propagation



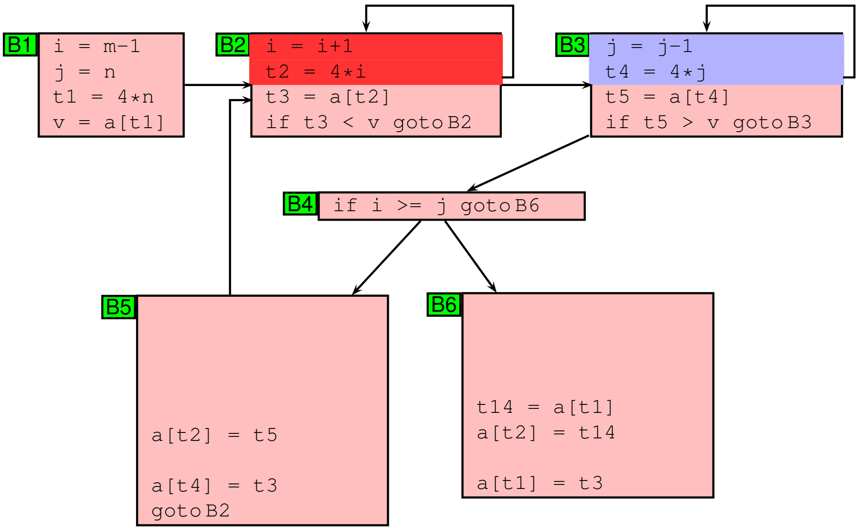
Copy Propagation



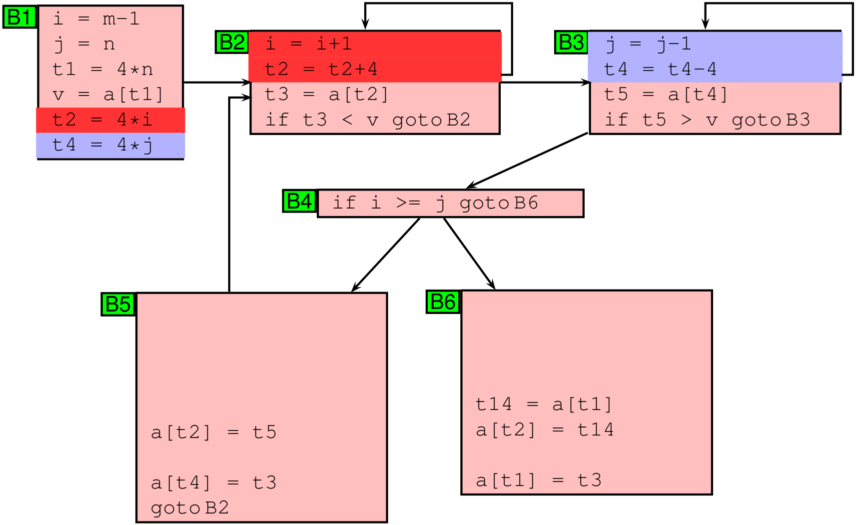
Copy Propagation



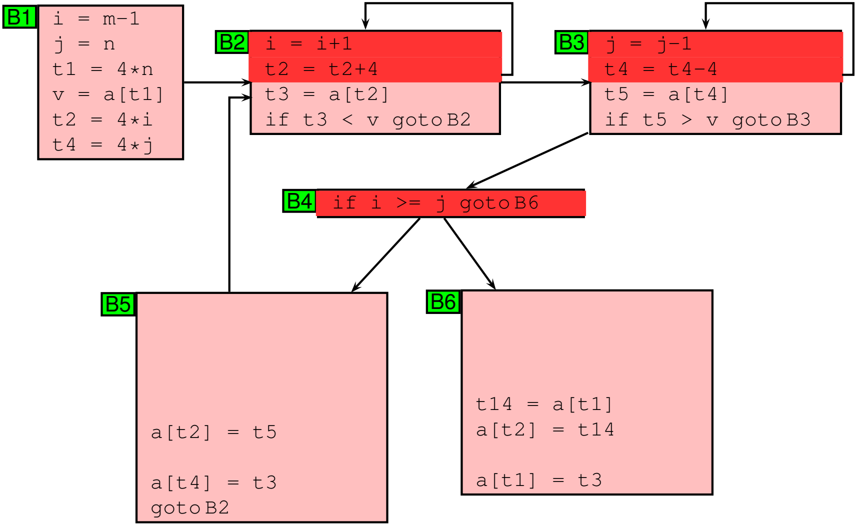
Strength Reduction



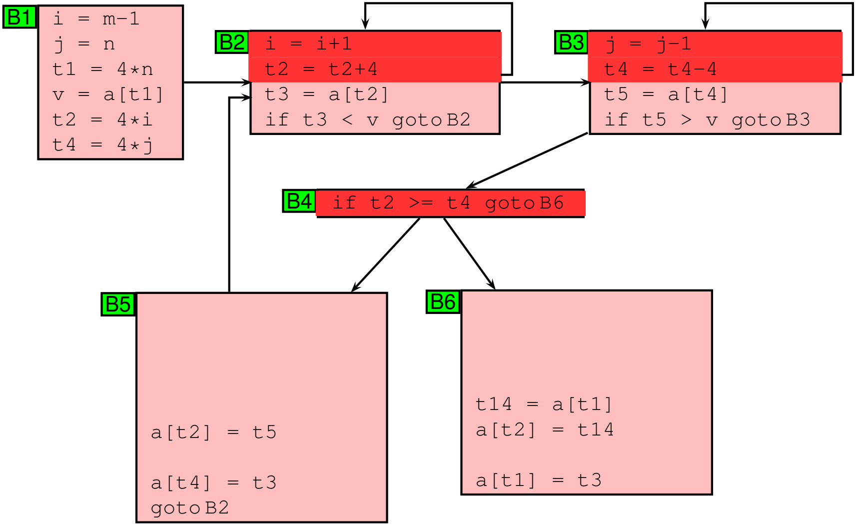
Strength Reduction



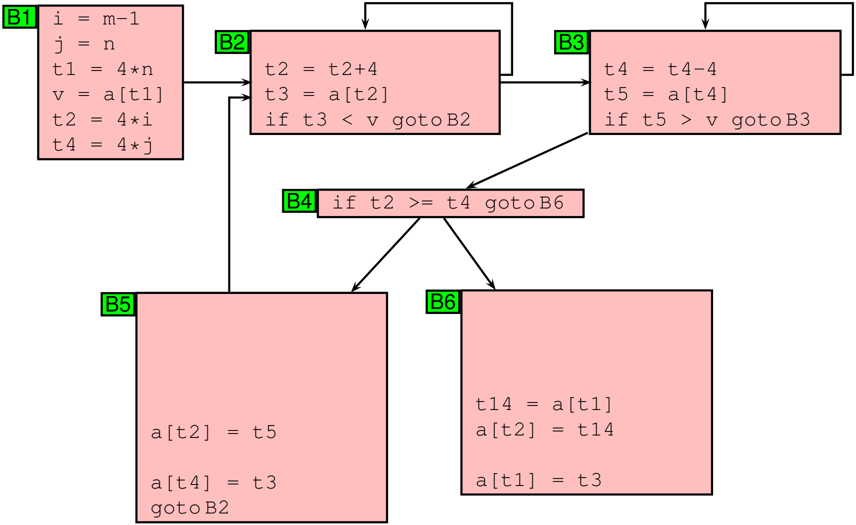
Induction Variable Elimination



Induction Variable Elimination



Dead Code Elimination (Again!)



Benefits

B#	# Stmts before Opts	# Stmts after Opts
B1	4	6
B2	4	3
B3	4	3
B4	1	1
B5	9	3
B6	8	3

- **Assumptions:**

Benefits

B#	# Stmts before Opts	# Stmts after Opts
B1	4	6
B2	4	3
B3	4	3
B4	1	1
B5	9	3
B6	8	3

- **Assumptions:**
  - Unit cost for each stmt

Benefits

B#	# Stmts before Opts	# Stmts after Opts
B1	4	6
B2	4	3
B3	4	3
B4	1	1
B5	9	3
B6	8	3

- **Assumptions:**
  - Unit cost for each stmt
  - Outer loop: 10 iterations

Benefits

B#	# Stmts before Opts	# Stmts after Opts
B1	4	6
B2	4	3
B3	4	3
B4	1	1
B5	9	3
B6	8	3

- **Assumptions:**
  - Unit cost for each stmt
  - Outer loop: 10 iterations
  - Inner loops: 100 iterations each

Benefits

B#	# Stmts before Opts	# Stmts after Opts
B1	4	6
B2	4	3
B3	4	3
B4	1	1
B5	9	3
B6	8	3

- **Assumptions:**
  - Unit cost for each stmt
  - Outer loop: 10 iterations
  - Inner loops: 100 iterations each
- **Cost of Execution:**

Benefits

B#	# Stmts before Opts	# Stmts after Opts
B1	4	6
B2	4	3
B3	4	3
B4	1	1
B5	9	3
B6	8	3

- **Assumptions:**
  - Unit cost for each stmt
  - Outer loop: 10 iterations
  - Inner loops: 100 iterations each
- **Cost of Execution:**
  - **Original Program:**

    1*4 + 100*4 + 100*4 + 10*1 + 10*9 + 1*8 = 912

Benefits

B#	# Stmts before Opts	# Stmts after Opts
B1	4	6
B2	4	3
B3	4	3
B4	1	1
B5	9	3
B6	8	3

- **Assumptions:**
  - Unit cost for each stmt
  - Outer loop: 10 iterations
  - Inner loops: 100 iterations each
- **Cost of Execution:**
  - **Original Program:**
    1*4 + 100*4 + 100*4 + 10*1 + 10*9 + 1*8 = 912
  - **Optimized Program:**
    1*6 + 100*3 + 100*3 + 10*1 + 10*3 + 1*3 = 649

Machine Dependent Optimizations

Peephole Optimizations

- Target code often contains redundant instructions and suboptimal constructs

Peephole Optimizations

- Target code often contains redundant instructions and suboptimal constructs
- Examine a short sequence of target instruction (peephole) and replace by a shorter or faster sequence

Peephole Optimizations

- Target code often contains redundant instructions and suboptimal constructs
- Examine a short sequence of target instruction (peephole) and replace by a shorter or faster sequence
- Peephole is a small moving window on the target systems

Peephole Optimizations: Examples

- Redundant loads and stores

Peephole Optimizations: Examples

- Redundant loads and stores
- Consider the code sequence

$$\texttt{move}\ R_0,\ a$$
$$\texttt{move}\ a,\ R_0$$

Peephole Optimizations: Examples

- Redundant loads and stores
- Consider the code sequence

$$\begin{array}{l}\texttt{move}\ \ R_0\texttt{,}\ \ a \\ \texttt{move}\ \ a\texttt{,}\ \ R_0\end{array}$$

- Is instruction 2 redundant? Can we always remove it?

Peephole Optimizations: Examples

- Redundant loads and stores
- Consider the code sequence

```
move R0, a
move a, R0
```

- Is instruction 2 redundant? Can we always remove it?
  - YES, if it does not have label

Peephole Optimizations: Unreachable code

- Consider the following code

```
int debug = 0;
if (debug) {
    print debugging info
}
```

Peephole Optimizations: Unreachable code

- Consider the following code

```
int debug = 0;
if (debug) {
    print debugging info
}
```

- This may be translated as

```
    int debug = 0;
    if (debug == 1) goto L1
    goto L2
L1: print debugging info
L2:
```

Peephole Optimizations: Unreachable code

- Eliminate Jumps

```
     int debug = 0;
     if (debug != 1) goto L2
     print debugging info
L2:
```

Peephole Optimizations: Unreachable code

- Eliminate Jumps

```
    int debug = 0;
    if (debug != 1) goto L2
    print debugging info
L2:
```

- Constant propagation

```
    int debug = 0;
    if (0 != 1) goto L2
    print debugging info
L2:
```

Peephole Optimizations: Unreachable code

- Constant folding and simplification: Since `if` condition is always true, the code becomes:

```
    goto L2
    print debugging info
L2:
```

Peephole Optimizations: Unreachable code

- Constant folding and simplification: Since `if` condition is always true, the code becomes:

```
    goto L2
    print debugging info
L2:
```

- The print statement is now unreachable. Therefore, the code becomes

```
L2:
```

Peephole Optimizations: Jump Optimizations

- Replace jump-over-jumps

```
    goto L1
      ⋮
L1: goto L2
```

Peephole Optimizations: Jump Optimizations

- Replace jump-over-jumps

goto L1 ⋮ L1: goto L2	can be replaced by	goto L2 ⋮ L1: goto L2

Peephole Optimizations: Simplify Algebraic Expressions

- Remove

```
x = x + 0;
x = x * 1;
```

Peephole Optimizations: Strength Reduction

- Replace $X^\wedge 2$ by $X * X$

Peephole Optimizations: Strength Reduction

- Replace $X^\wedge 2$ by $X * X$
- Replace multiplication by left shift

Peephole Optimizations: Strength Reduction

- Replace $X^\wedge 2$ by $X * X$
- Replace multiplication by left shift
- Replace divison by right shift

Peephole Optimizations: Use of Faster Instructions

- Replace

  Add #1, R

  by

  Inc R

Course Logistics

Evaluation

- Assignments
- Course project
- Mid semester exam
- End semester exan
- Quizzes/Class participation
- Refer to course webpage for details.